

CSE 548 / AMS 542: Analysis of Algorithms

Prerequisites Review 7 (Dynamic Programming)

**Rezaul Chowdhury
Department of Computer Science
SUNY Stony Brook
Fall 2023**

The Rod Cutting Problem

Suppose you are given:

- a rod of length n inches, and
- a list of prices p_i for integer $i \in [1, n]$,
where p_i is the selling price of a rod of length i inches.

Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

The Rod Cutting Problem

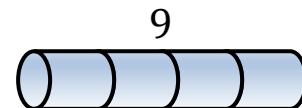
A sample price table for rods

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

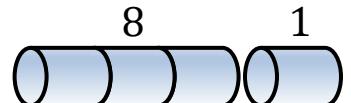
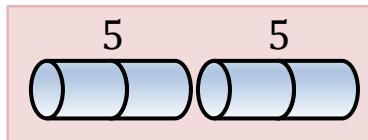
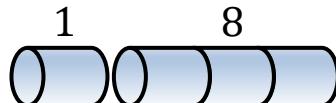
Solve the problem for $n = 4$ and the price table given above.

#pieces #ways

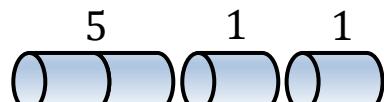
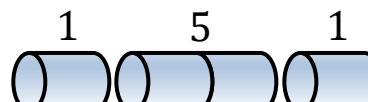
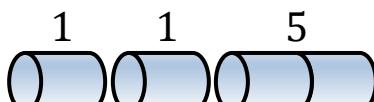
$$1 \quad \binom{3}{0} = 1$$



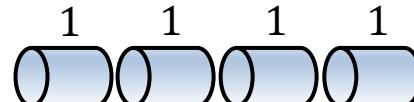
$$2 \quad \binom{3}{1} = 3$$



$$3 \quad \binom{3}{2} = 3$$



$$4 \quad \binom{3}{3} = 1$$



Total: $8 = 2^3$

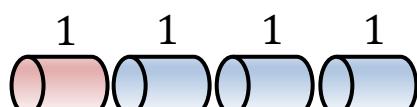
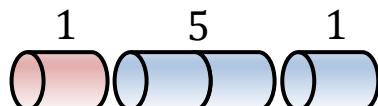
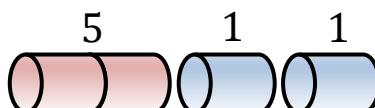
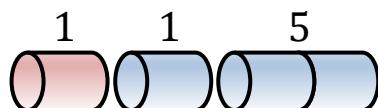
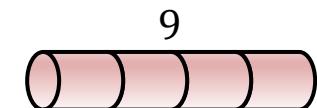
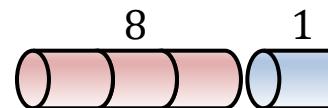
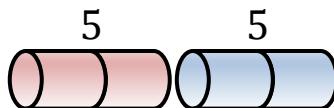
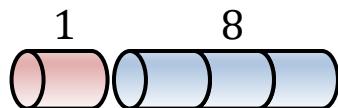
$$r_n = 5 + 5 = 10$$

Rod Cutting: Standard Recursive Algorithm

A sample price table for rods

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

There is a different way of looking at the cuts and thus computing r_n .



$$r_n = \begin{cases} 0, & \text{if } n = 0, \\ \max_{1 \leq i \leq n} \{p_i + r_{n-i}\}, & \text{if } n > 0. \end{cases}$$

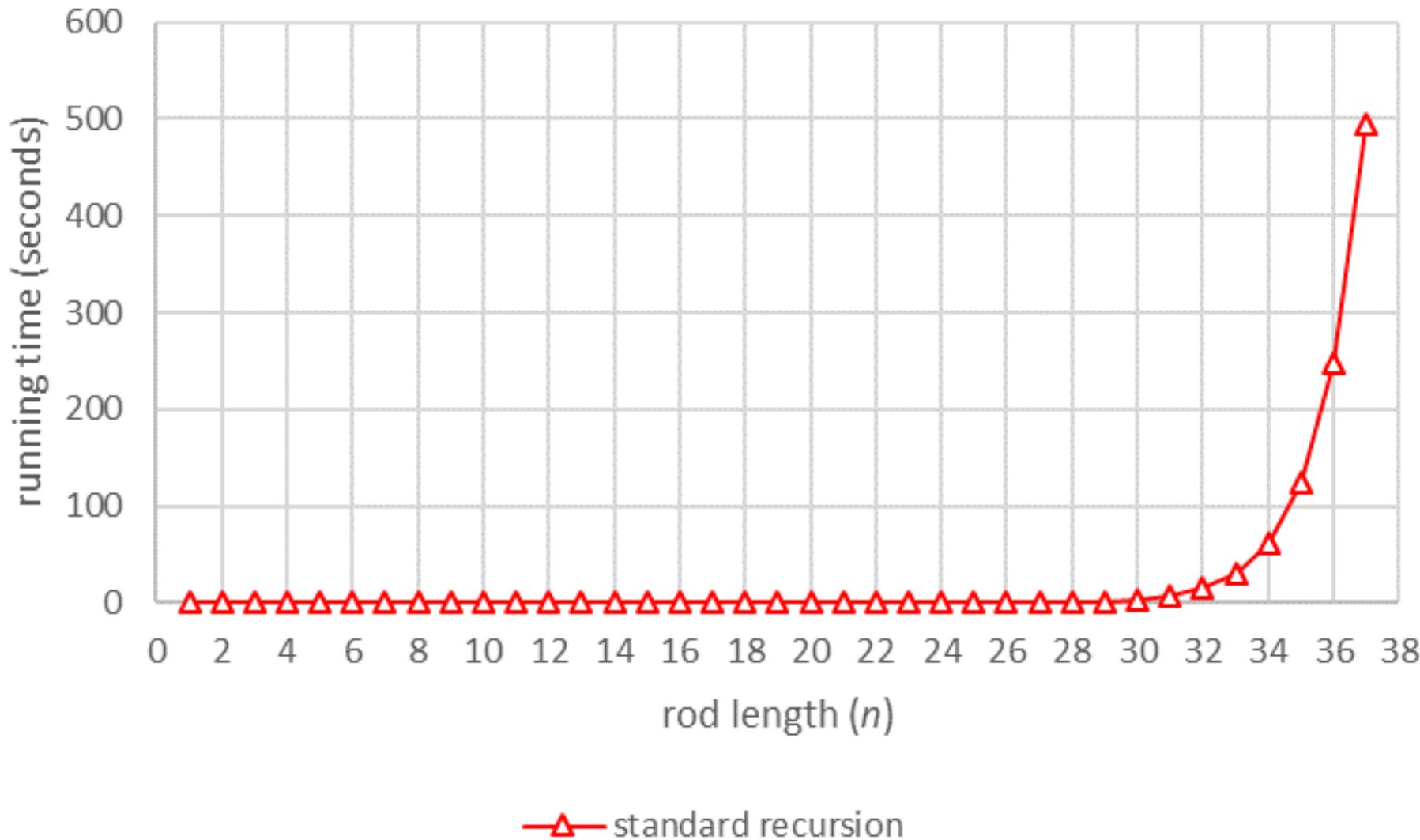
Rod Cutting: Standard Recursive Algorithm

$$r_n = \begin{cases} 0, & \text{if } n = 0, \\ \max_{1 \leq i \leq n} \{ p_i + r_{n-i} \}, & \text{if } n > 0. \end{cases}$$

CUT-ROD (p, n)

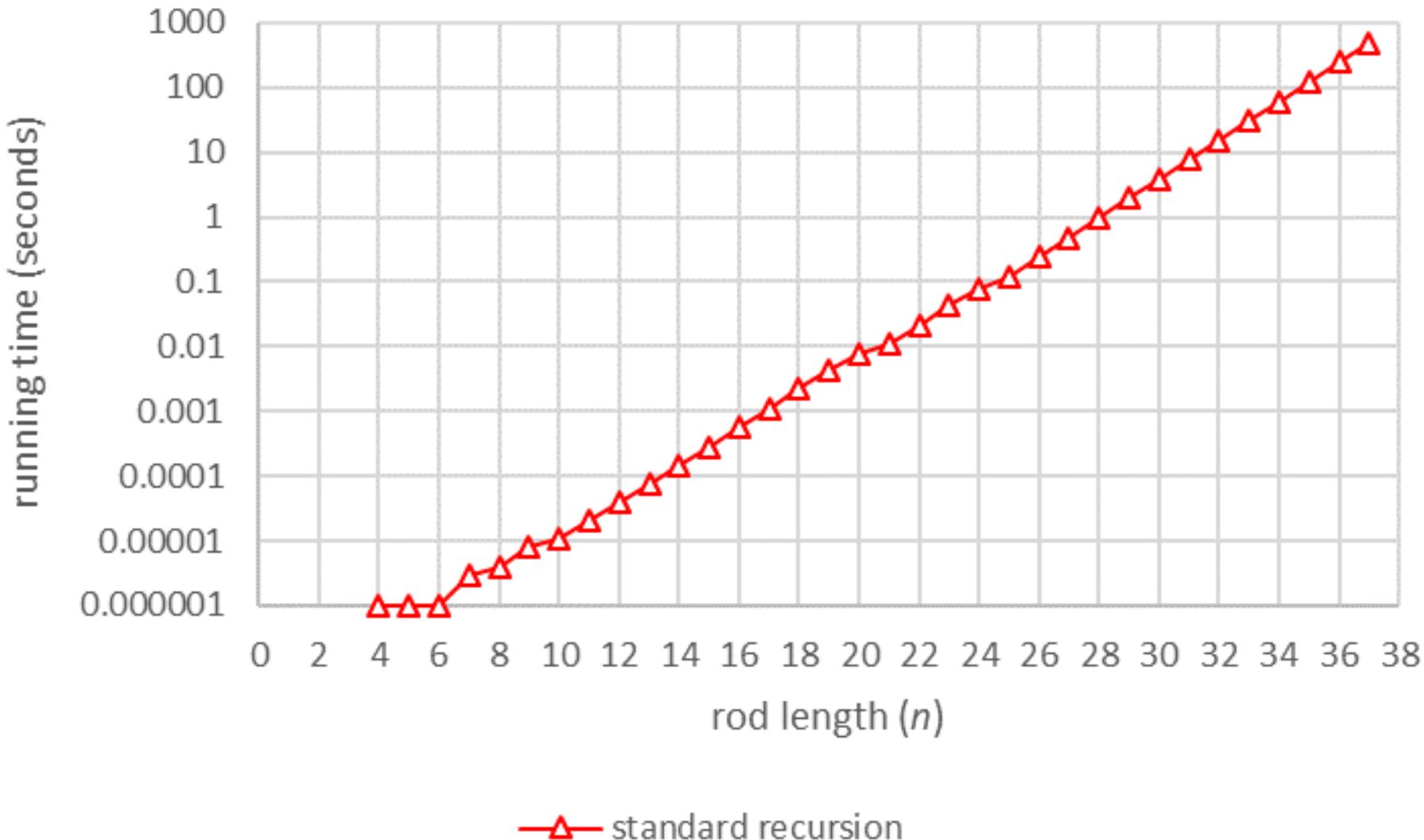
1. *if* $n = 0$ *then*
2. *return* 0
3. $q \leftarrow -\infty$
4. *for* $i \leftarrow 1$ *to* n *do*
5. $q \leftarrow \max\{ q, p[i] + \text{CUT-ROD} (p, n - i) \}$
6. *return* q

Rod Cutting: Standard Recursive Algorithm



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Standard Recursive Algorithm



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Standard Recursive Algorithm

CUT-ROD (p, n)

1. *if* $n = 0$ *then*
2. *return* 0
3. $q \leftarrow -\infty$
4. *for* $i \leftarrow 1$ *to* n *do*
5. $q \leftarrow \max\{ q, p[i] + \text{CUT-ROD} (p, n - i) \}$
6. *return* q

Let $T(n)$ be the running time of the algorithm on an input of size n .

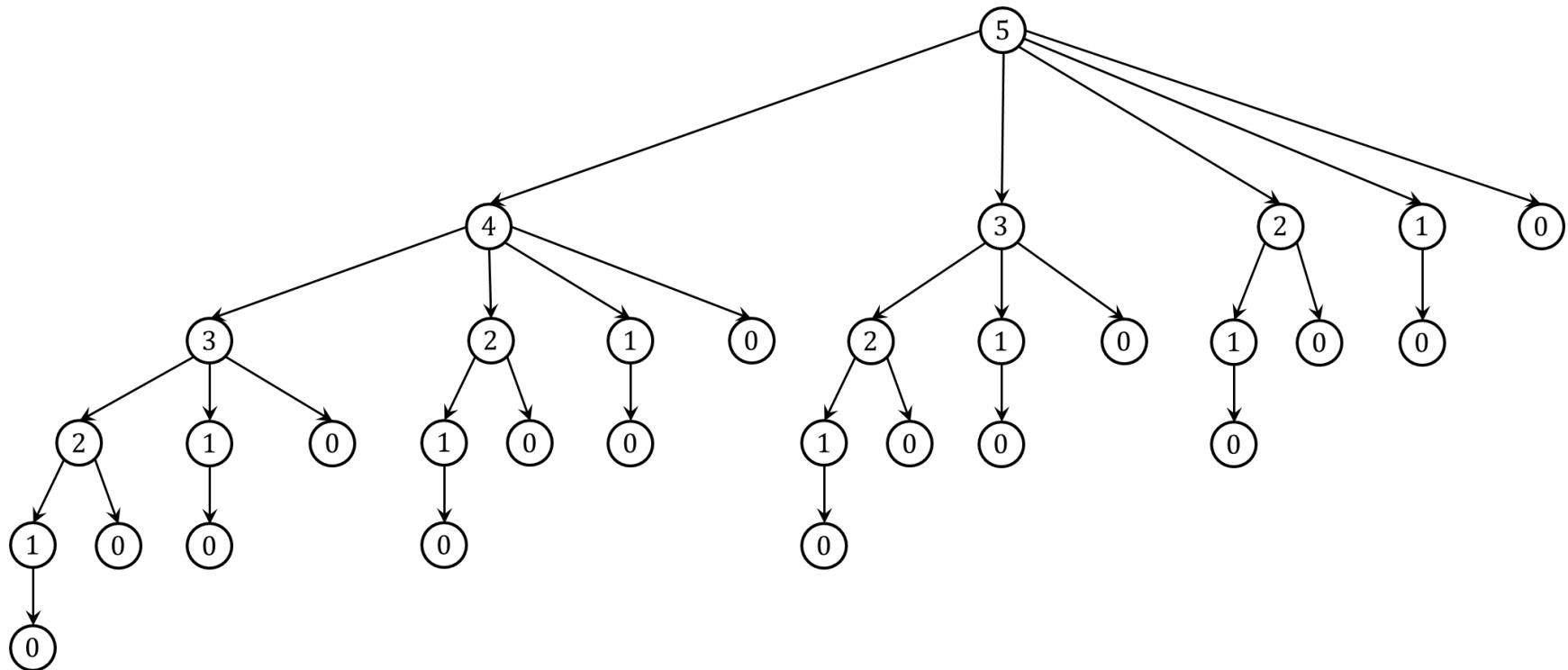
Then

$$T(n) = \begin{cases} \Theta(1), & \text{if } n = 0, \\ \sum_{i=1}^n T(n-i) + \Theta(1), & \text{if } n > 0. \end{cases}$$

Solving: $T(n) = \Theta(2^n)$.

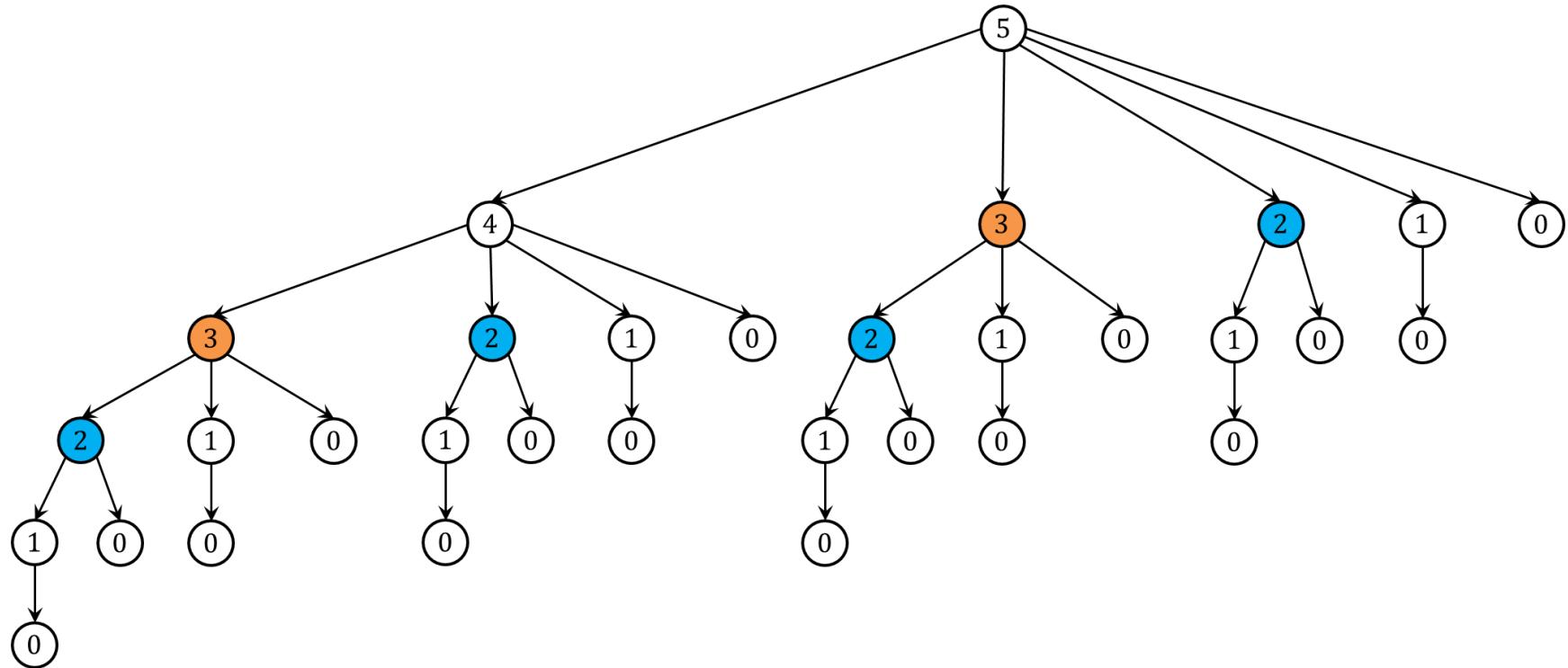
Rod Cutting: Standard Recursive Algorithm

When `CUT-ROD(n)` is called with $n = 5$, the values of n passed to the recursive function calls are shown below.



Rod Cutting: Standard Recursive Algorithm

When $\text{CUT-ROD}(n)$ is called with $n = 5$, the values of n passed to the recursive function calls are shown below.



We are calling $\text{CUT-ROD}(n)$ or solving the problem for the same value of n over and over again!

How about saving the solution when we solve the problem for any given value of n for the first time?

Rod Cutting: Recursion with Memoization (a.k.a. Backtracking with Memoization)

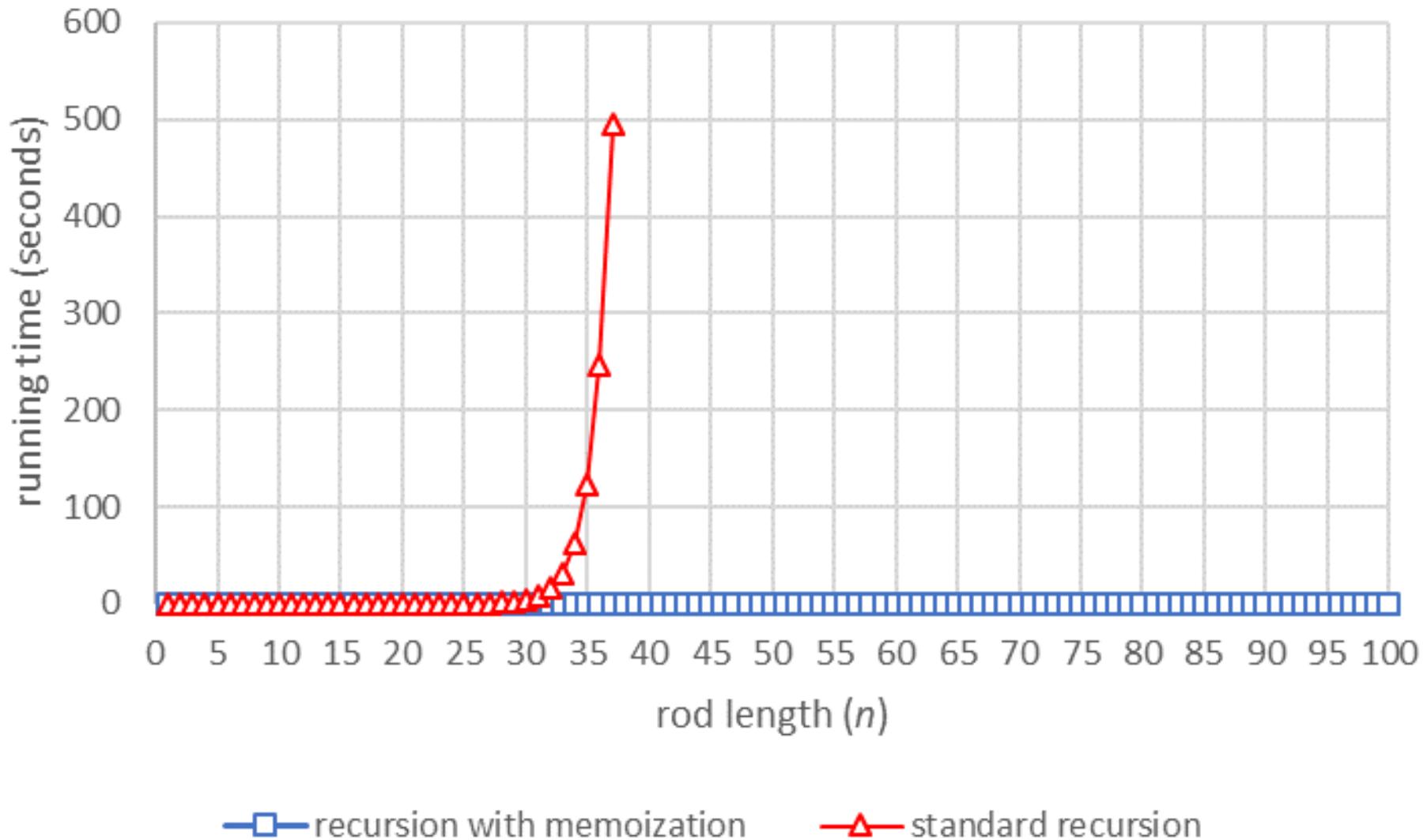
MEMOIZED-CUT-ROD (p, n)

1. $r[0..n] \leftarrow$ new array
2. *for* $i \leftarrow 0$ *to* n *do*
3. $r[i] \leftarrow -\infty$
4. *return* *MEMOIZED-CUT-ROD-AUX (p, n, r)*

MEMOIZED-CUT-ROD-AUX (p, n, r)

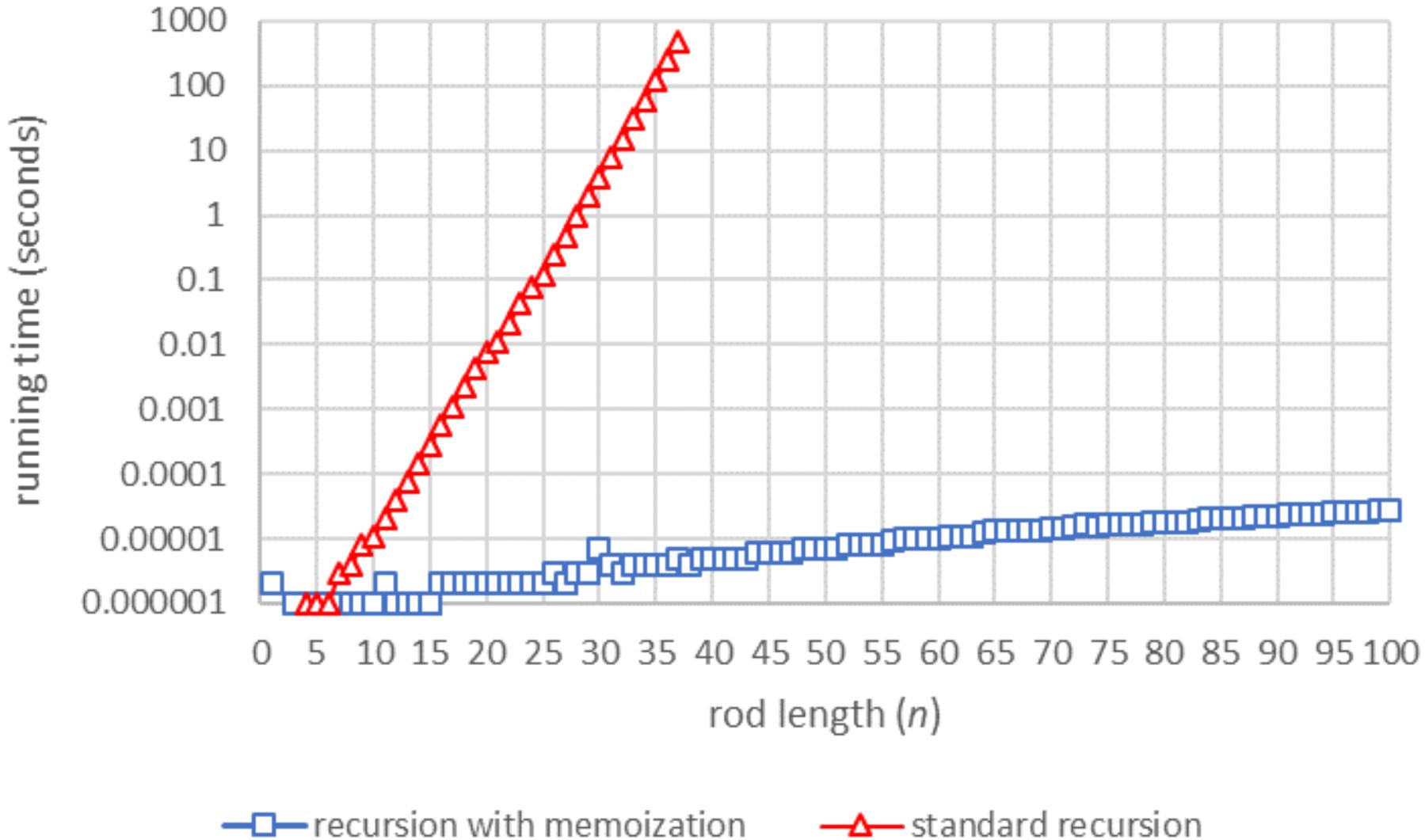
1. *if* $r[n] \geq 0$ *then*
2. *return* $r[n]$
3. *if* $n = 0$ *then*
4. $q \leftarrow 0$
3. *else* $q \leftarrow -\infty$
4. *for* $i \leftarrow 1$ *to* n *do*
5. $q \leftarrow \max\{ q, p[i] + \text{MEMOIZED-CUT-ROD-AUX} (p, n - i, r) \}$
6. $r[n] \leftarrow q$
7. *return* q

Rod Cutting: Recursion with Memoization



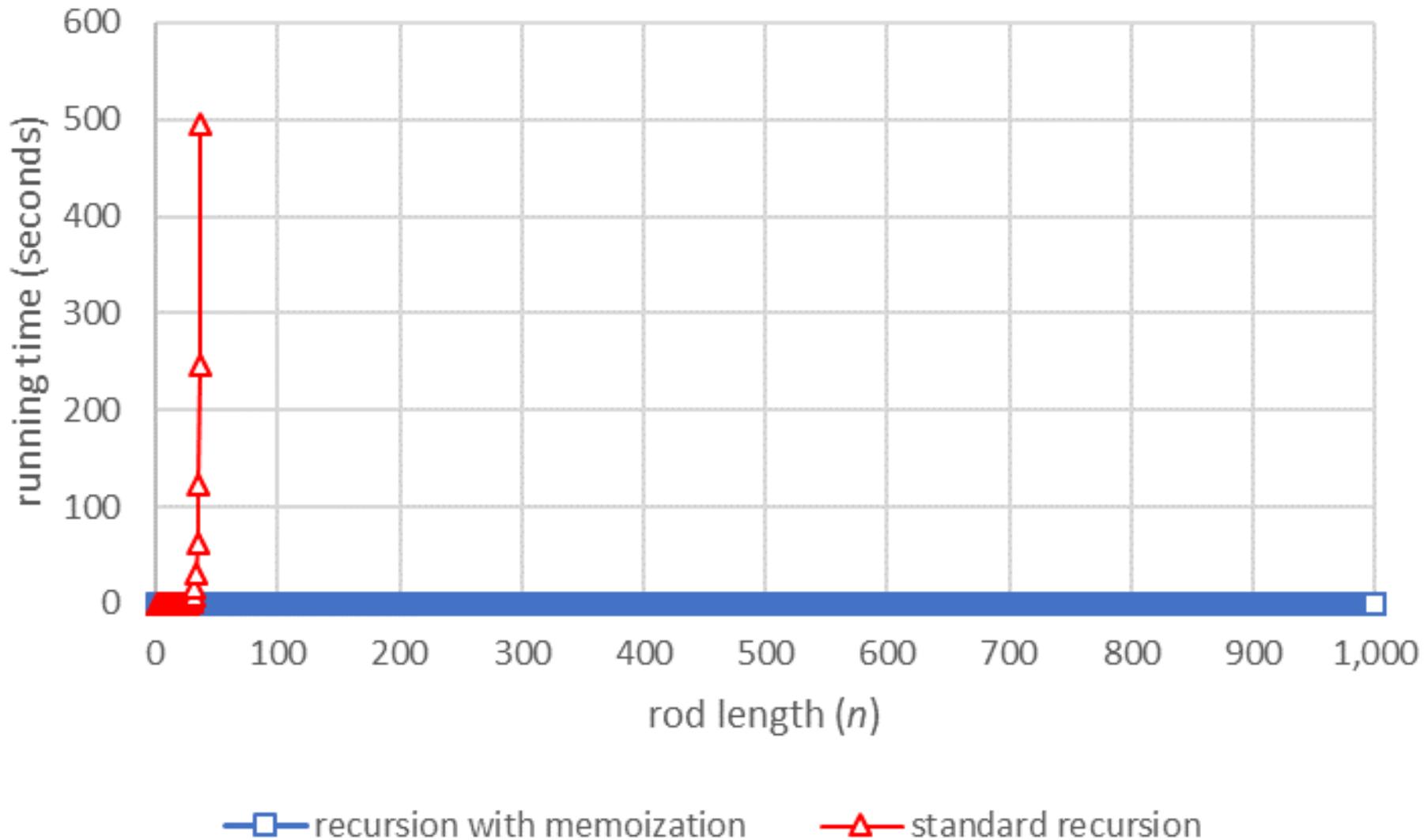
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



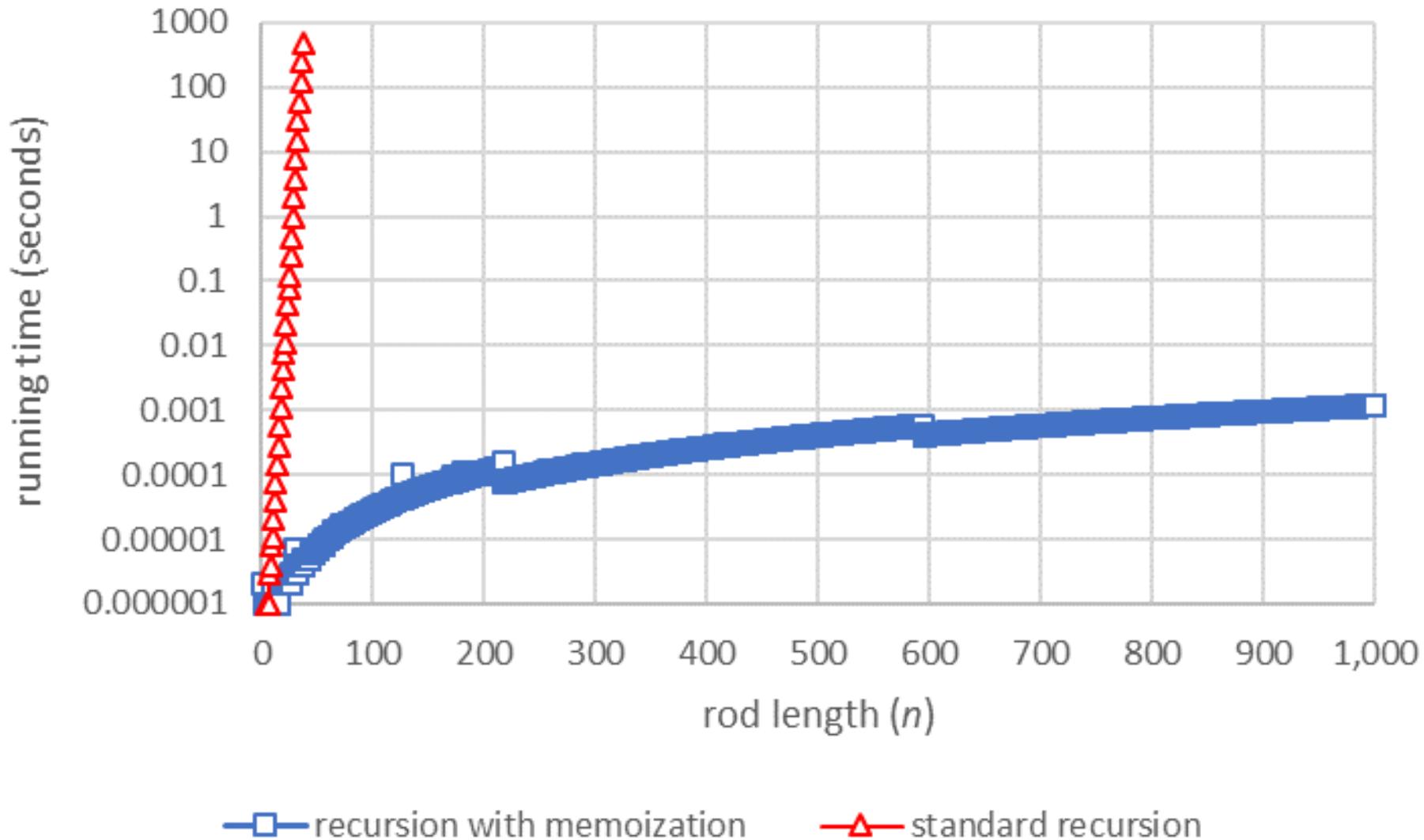
*Run on a dual-socket (2×8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



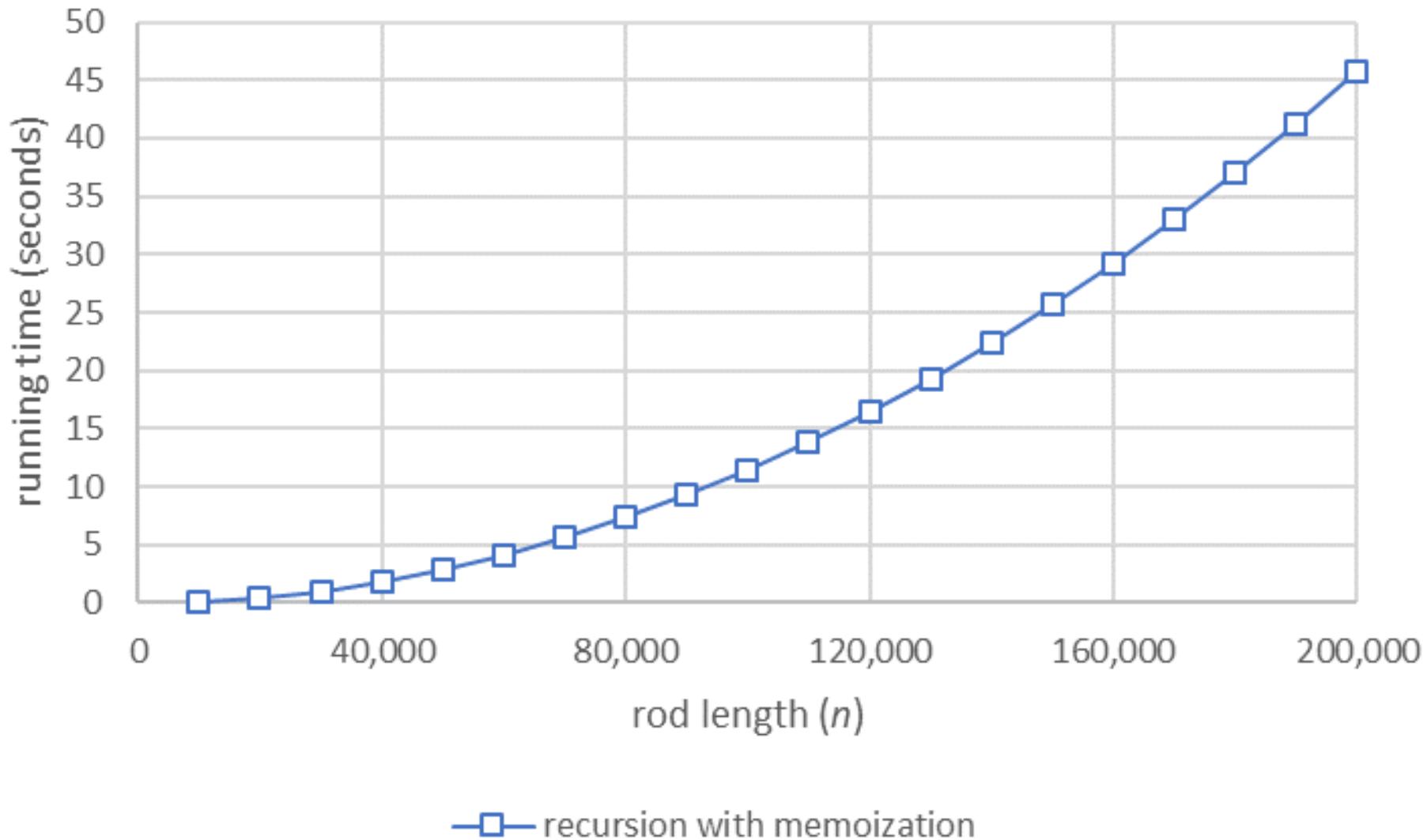
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursion with Memoization



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Bottom-up Dynamic Programming

BOTTOM-UP-CUT-ROD (p, n)

1. $r[0..n] \leftarrow$ new array
2. $r[0] \leftarrow 0$
3. *for* $j \leftarrow 1$ *to* n *do*
4. $q \leftarrow -\infty$
5. *for* $i \leftarrow 1$ *to* j *do*
6. $q \leftarrow \max\{ q, p[i] + r[j-i] \}$
7. $r[j] \leftarrow q$
8. *return* $r[n]$

Rod Cutting: Bottom-up Dynamic Programming

BOTTOM-UP-CUT-ROD (p, n)

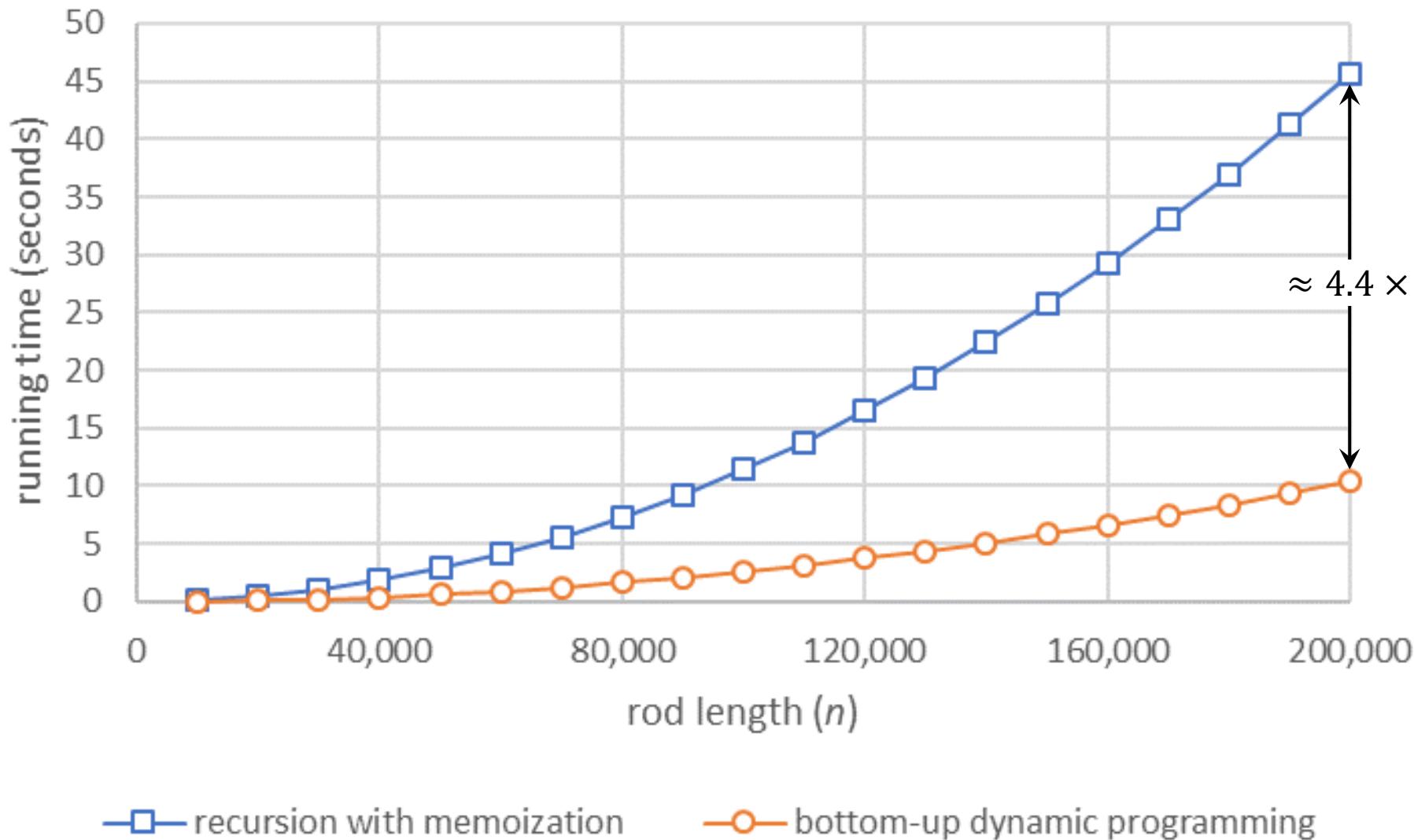
1. $r[0..n] \leftarrow$ new array
2. $r[0] \leftarrow 0$
3. *for* $j \leftarrow 1$ *to* n *do*
4. $q \leftarrow -\infty$
5. *for* $i \leftarrow 1$ *to* j *do*
6. $q \leftarrow \max\{ q, p[i] + r[j-i] \}$
7. $r[j] \leftarrow q$
8. *return* $r[n]$

Let $T(n)$ be the running time of *BOTTOM-UP-CUT-ROD* on an input of size n . Then

$$T(n) = O\left(1 + \sum_{j=1}^n \left(1 + \sum_{i=1}^j 1\right)\right) = O(n^2).$$

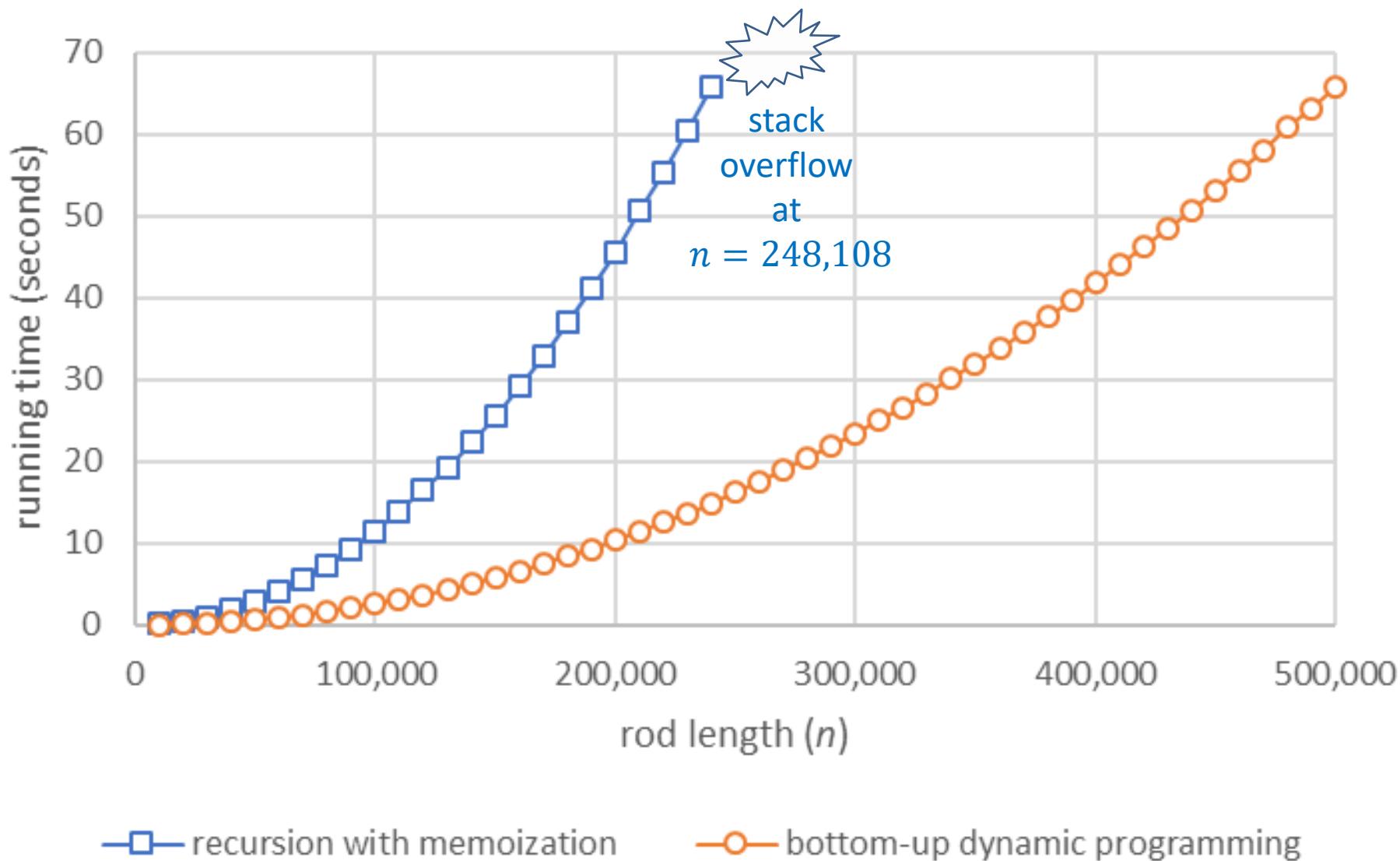
Note that *MEMOIZED-CUT-ROD* has the same $O(n^2)$ running time on an input of size n .

Rod Cutting: Bottom-up Dynamic Programming



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Bottom-up Dynamic Programming



*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Extracting the Solution

EXTENDED-BOTTOM-UP-CUT-ROD (p, n)

1. $r[0..n] \leftarrow$ new array, $s[0..n] \leftarrow$ new array
2. $r[0] \leftarrow 0$
3. *for* $j \leftarrow 1$ *to* n *do*
4. $q \leftarrow -\infty$
5. *for* $i \leftarrow 1$ *to* j *do*
6. *if* $q < p[i] + r[j-i]$ *then*
7. $q \leftarrow p[i] + r[j-i]$
8. $s[j] \leftarrow i$
9. $r[j] \leftarrow q$
10. *return* r and s

PRINT-CUT-ROD-SOLUTION (p, n)

1. $(r, s) \leftarrow$ *EXTENDED-BOTTOM-UP-CUT-ROD (p, n)*
2. *while* $n > 0$ *do*
3. *print* $s[n]$
4. $n \leftarrow n - s[n]$

Rod Cutting: Extracting the Solution

A sample price table for rods

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

EXTENDED-BOTTOM-UP-CUT-ROD(p, n) returns the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

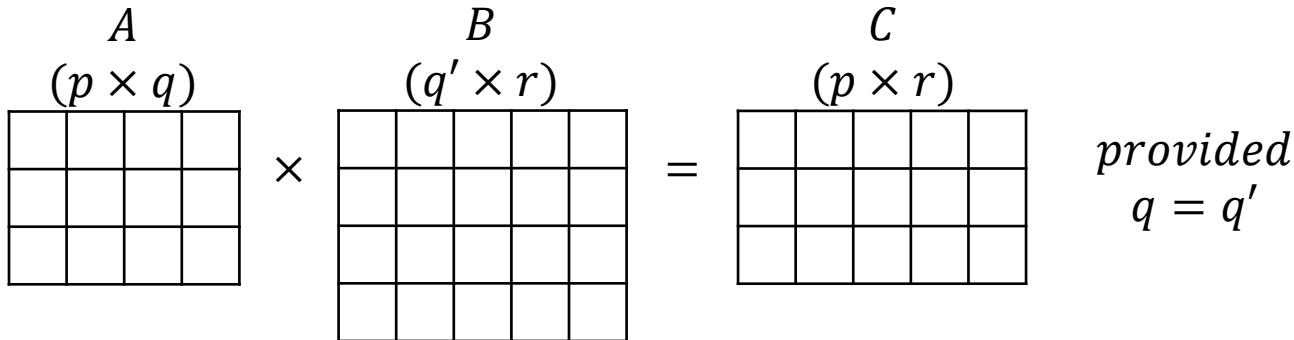
Matrix-Chain Multiplication

$$\begin{array}{c} A \\ (p \times q) \end{array} \quad \times \quad \begin{array}{c} B \\ (q \times r) \end{array} \quad = \quad \begin{array}{c} C \\ (p \times r) \end{array}$$

A $p \times q$ matrix A and a $q' \times r$ matrix B can be multiplied provided $q = q'$.

The result will be a $p \times r$ matrix C .

Matrix-Chain Multiplication



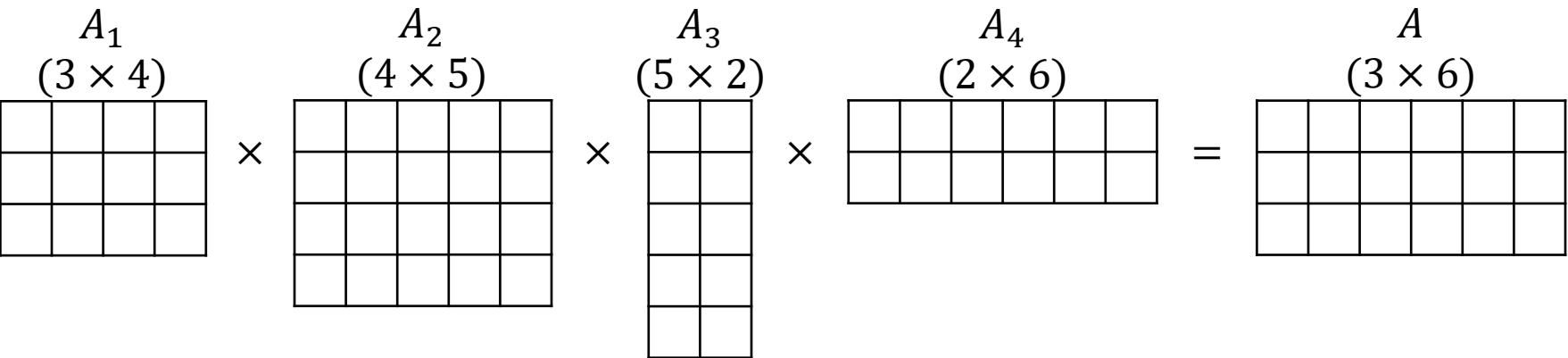
MATRIX-MULTIPLY (p, q, A, q', r, B)

1. *if* $q \neq q'$ *then*
2. *error* “incompatible dimensions”
3. *else*
4. $C \leftarrow$ new $p \times r$ matrix
5. *for* $i \leftarrow 1$ *to* p *do*
6. *for* $j \leftarrow 1$ *to* r *do*
7. $C[i,j] \leftarrow 0$
8. *for* $k \leftarrow 1$ *to* q *do*
9. $C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$
10. *return* C

Time needed to multiply the $p \times q$ matrix A and the $q \times r$ matrix B is dominated by the total number pqr of scalar multiplications performed in line 7.

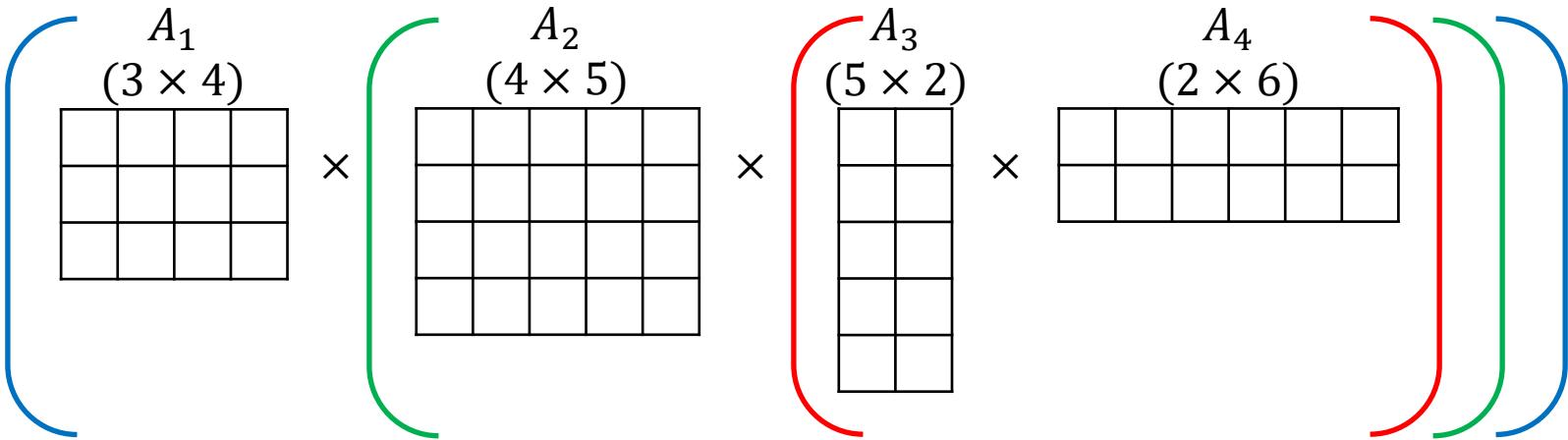
Hence, running time of the algorithm is $\Theta(pqr)$.

Matrix-Chain Multiplication

$$\begin{array}{c} A_1 \\ (3 \times 4) \end{array} \times \begin{array}{c} A_2 \\ (4 \times 5) \end{array} \times \begin{array}{c} A_3 \\ (5 \times 2) \end{array} \times \begin{array}{c} A_4 \\ (2 \times 6) \end{array} = \begin{array}{c} A \\ (3 \times 6) \end{array}$$


We can multiply the four matrices on the left hand side in five distinct orders.

Matrix-Chain Multiplication

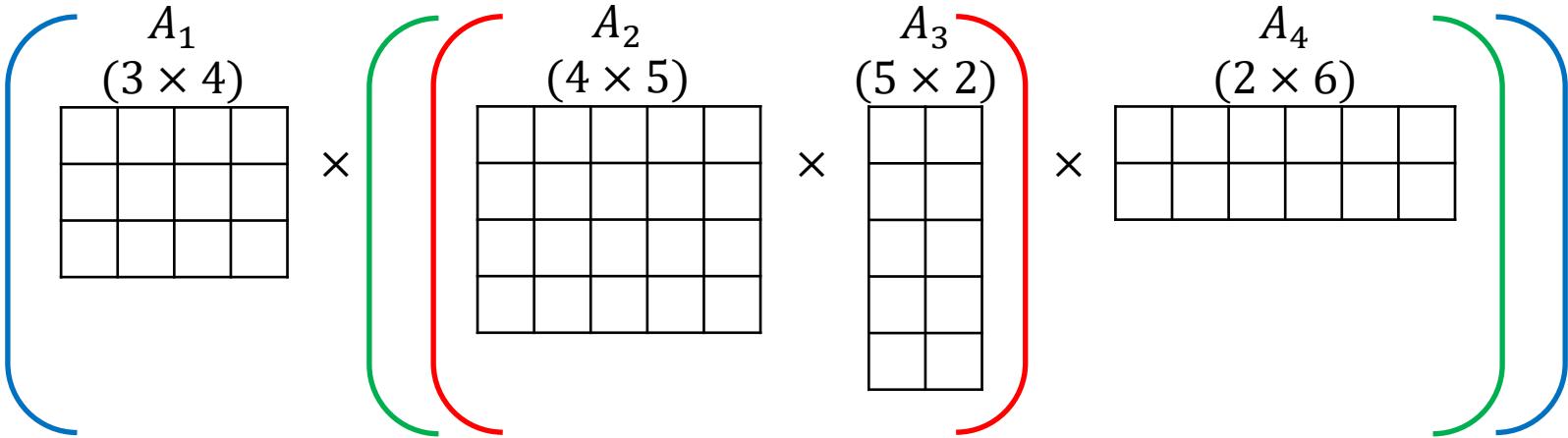


number of scalar multiplications

$$= 5 \times 2 \times 6 + 4 \times 5 \times 6 + 3 \times 4 \times 6$$

$$= 252$$

Matrix-Chain Multiplication

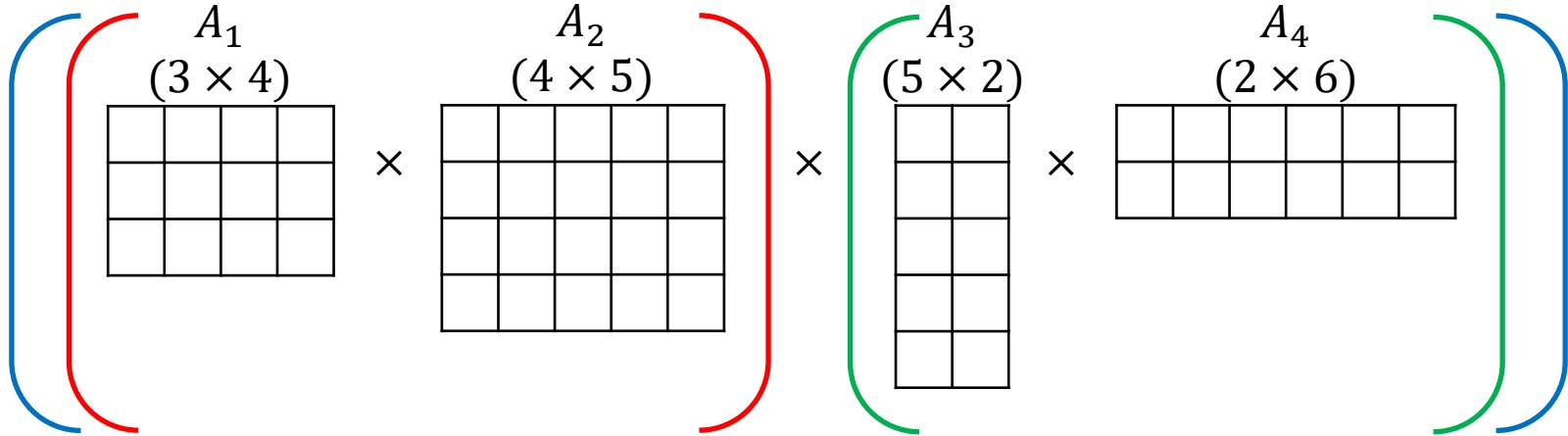


number of scalar multiplications

$$= 4 \times 5 \times 2 + 4 \times 2 \times 6 + 3 \times 4 \times 6$$

$$= 160$$

Matrix-Chain Multiplication

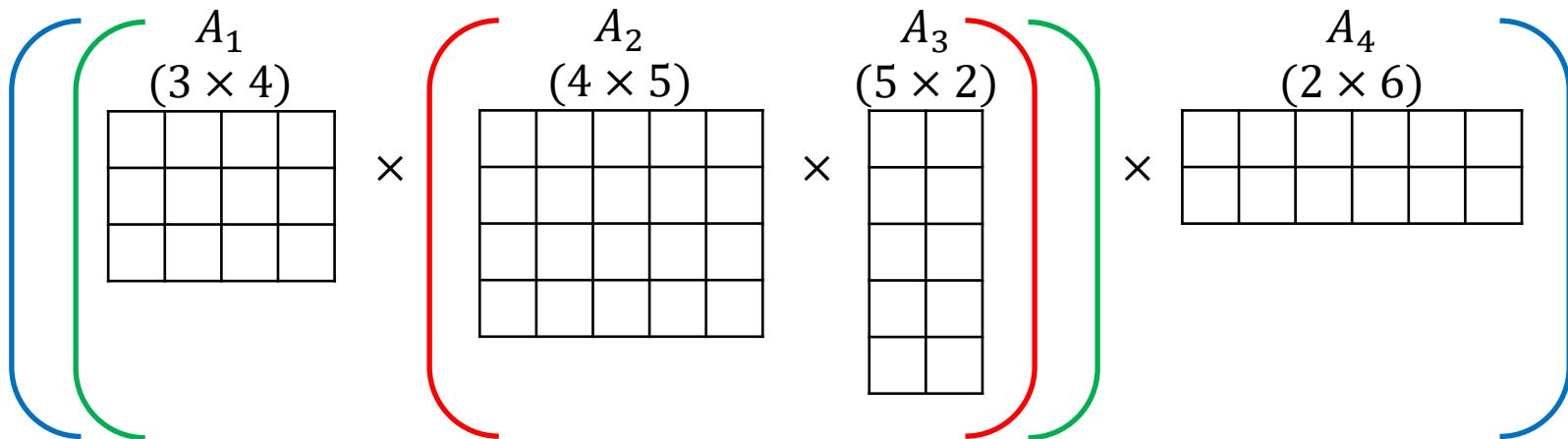


number of scalar multiplications

$$= 3 \times 4 \times 5 + 5 \times 2 \times 6 + 3 \times 5 \times 6$$

$$= 210$$

Matrix-Chain Multiplication

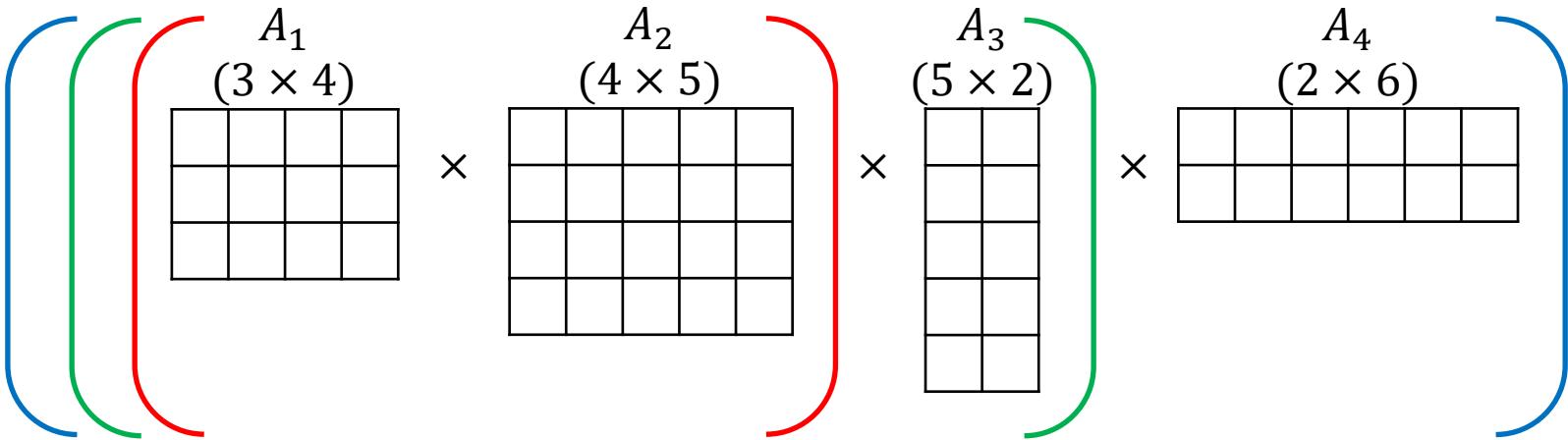


number of scalar multiplications

$$= 4 \times 5 \times 2 + 3 \times 4 \times 2 + 3 \times 2 \times 6$$

$$= 100$$

Matrix-Chain Multiplication



number of scalar multiplications

$$= 3 \times 4 \times 5 + 3 \times 5 \times 2 + 3 \times 2 \times 6$$

$$= 126$$

Matrix-Chain Multiplication

The matrix-chain multiplication problem:

Given a chain $\langle A_1, A_2, \dots, A_n \rangle$ of n matrices,

where for $i = 1, 2, \dots, n$, matrix A_i has dimension $p_{i-1} \times p_i$,
fully parenthesize the product $A_1 A_2 \dots A_n$

in a way that minimizes the number of scalar multiplications.

Matrix-Chain Multiplication

Let $P(n)$ = number of parenthesizations of a sequence of n matrices.

Then

$$P(n) = \begin{cases} 1, & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n - k), & \text{if } n \geq 2. \end{cases}$$

Very easy to show that $P(n) = \Omega(2^n)$.

Hence, exhaustively checking all possible parenthesizations of the given chain of matrices does not give an efficient algorithm.

Matrix-Chain Mult: Standard Recursive Algorithm

Let $A_{i\dots j} = A_i A_{i+1} \dots A_{j-1} A_j$ for $1 \leq i \leq j \leq n$.

Let $m(i, j)$ = the minimum number of scalar multiplications needed
to compute the matrix $A_{i\dots j}$.

Then $m(1, n)$ = the minimum number of scalar multiplications
needed to compute $A_{1\dots n}$ (i.e., solve the entire
problem).

$$m(i, j) = \begin{cases} 0, & \text{if } i = j, \\ \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1} p_k p_j\}, & \text{if } i < j. \end{cases}$$

Matrix-Chain Mult: Standard Recursive Algorithm

RECURSIVE-MATRIX-CHAIN (p, i, j)

1. *if* $i = j$ *then*

2. *return* 0

3. $q \leftarrow \infty$

4. *for* $k \leftarrow i$ *to* $j - 1$ *do*

5. $q \leftarrow \min \left(\begin{array}{c} q, \\ \text{RECURSIVE-MATRIX-CHAIN (} p, i, k \text{)} \\ + \text{RECURSIVE-MATRIX-CHAIN (} p, k + 1, j \text{)} \\ + p_{i-1}p_kp_j \end{array} \right)$

6. *return* q

Matrix-Chain Mult: Standard Recursive Algorithm

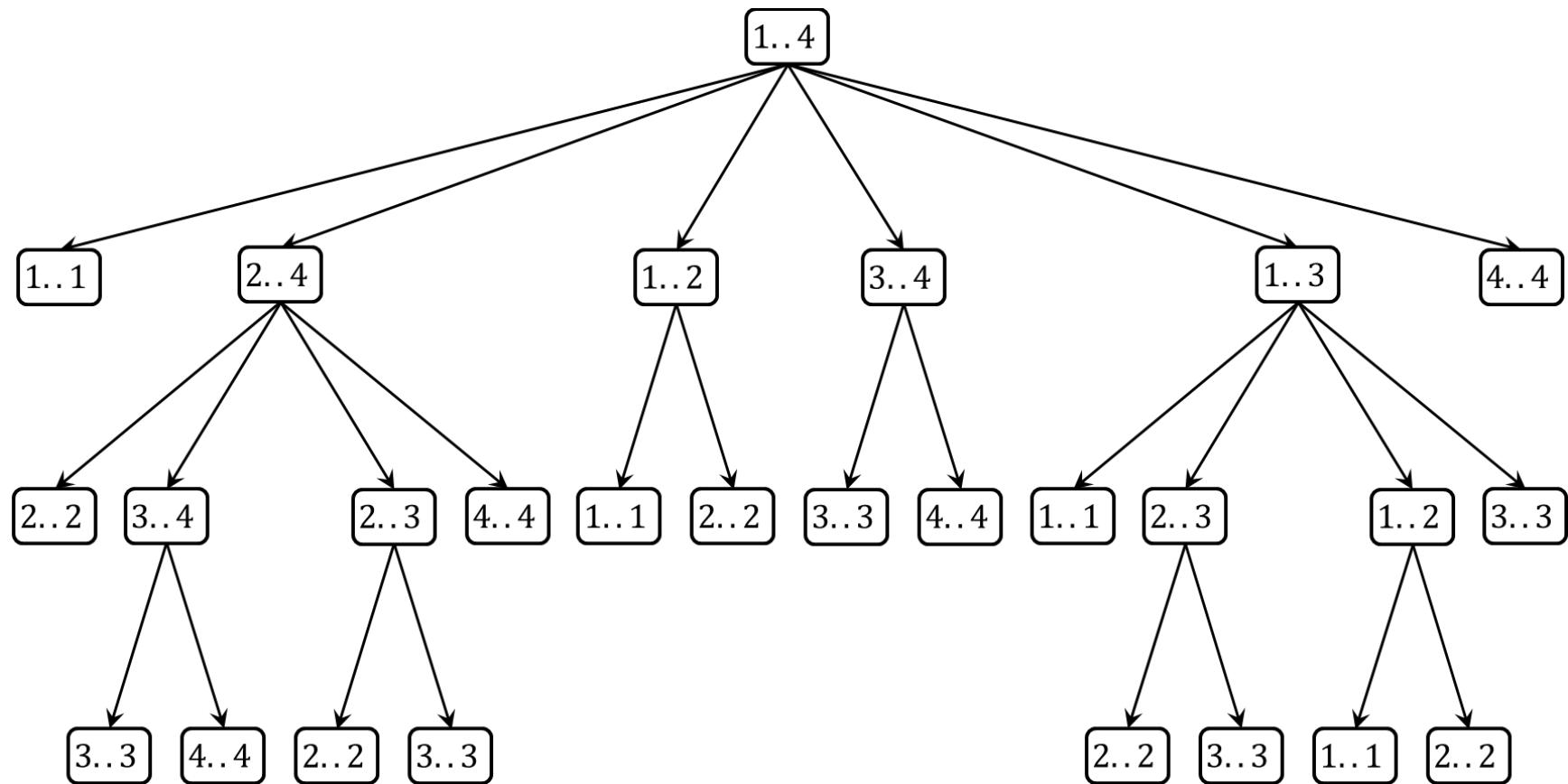
Let $T(n)$ be the running time of the algorithm on an input of size n .

Then

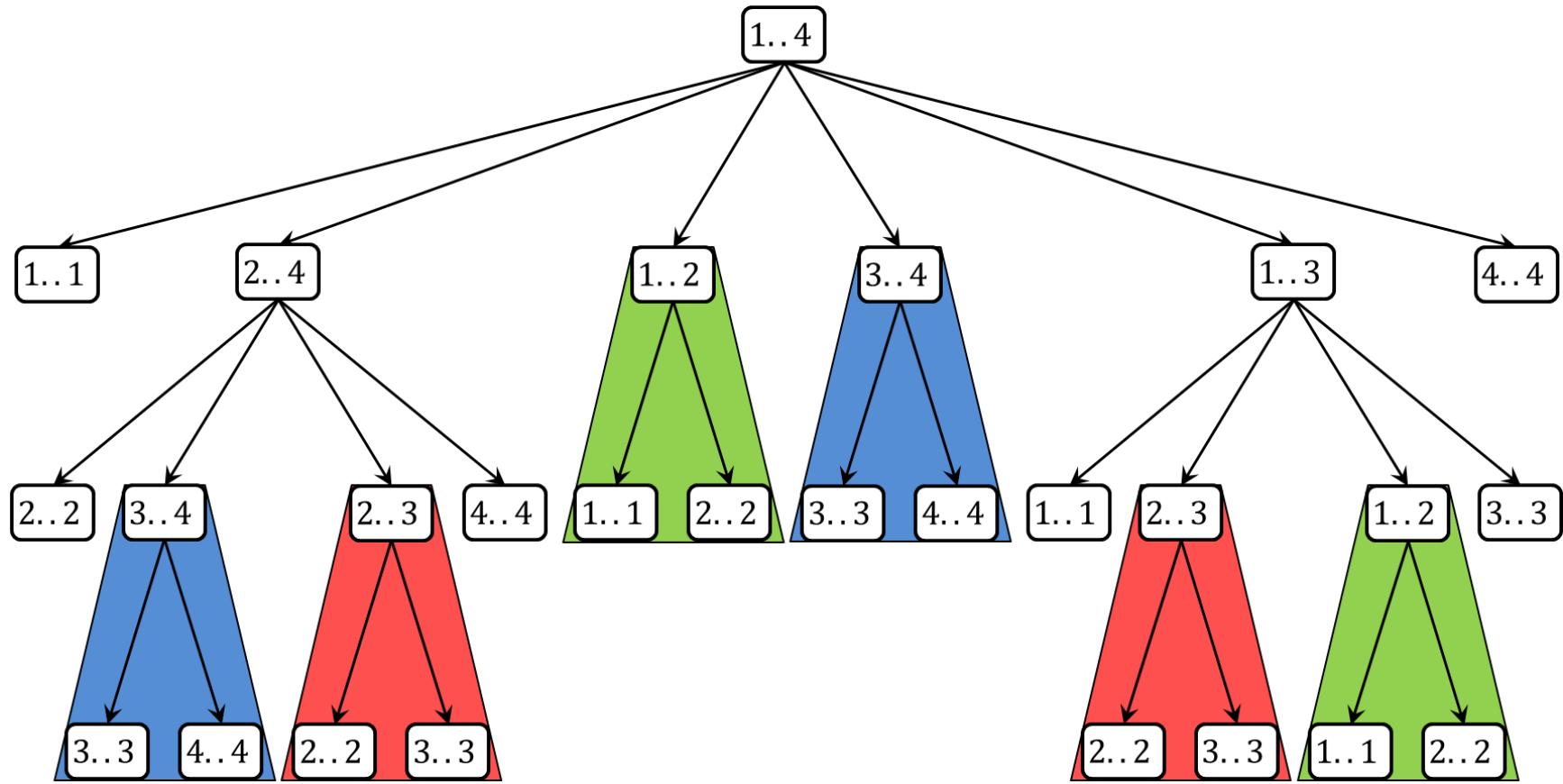
$$T(n) \geq \begin{cases} 1, & \text{if } n = 1, \\ 1 + \sum_{k=1}^{n-1} (T(k) + T(n - k) + 1), & \text{if } n > 1. \end{cases}$$

Solving: $T(n) \geq 2^{n-1} \Rightarrow T(n) = \Omega(2^n)$.

Matrix-Chain Mult: Standard Recursive Algorithm



Matrix-Chain Mult: Standard Recursive Algorithm



Matrix-Chain Mult: Recursion with Memoization

MEMOIZED-MATRIX-CHAIN (p)

1. $n \leftarrow p.length - 1$
2. $m[1..n, 1..n] \leftarrow \text{new table}$
3. *for* $i \leftarrow 1$ *to* n *do*
4. *for* $j \leftarrow i$ *to* n *do*
5. $m[i, j] \leftarrow \infty$
6. *return* *LOOKUP-CHAIN* ($m, p, 1, n$)

LOOKUP-CHAIN (m, p, i, j)

1. *if* $m[i, j] < \infty$ *then*
2. *return* $m[i, j]$
3. *if* $i = j$ *then*
4. $m[i, j] \leftarrow 0$
5. *for* $k \leftarrow i$ *to* $j - 1$ *do*
6. $q \leftarrow \text{LOOKUP-CHAIN} (m, p, i, k)$
 + *LOOKUP-CHAIN* ($m, p, k + 1, j$)
 + $p_{i-1}p_kp_j$
7. *if* $q < m[i, j]$ *then*
8. $m[i, j] \leftarrow q$
9. *return* $m[i, j]$

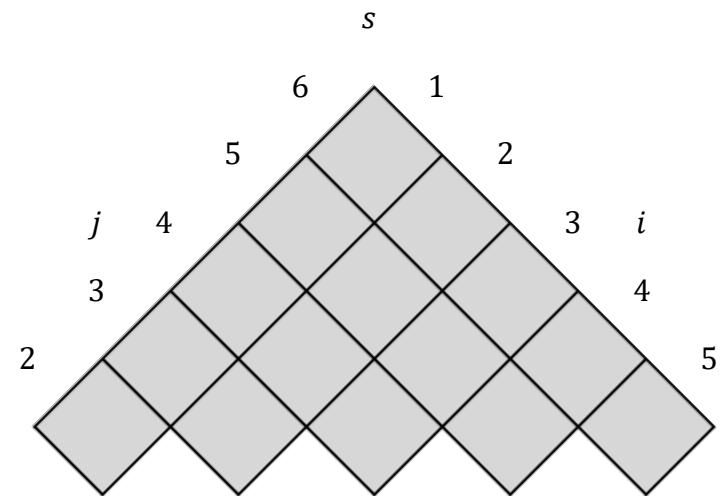
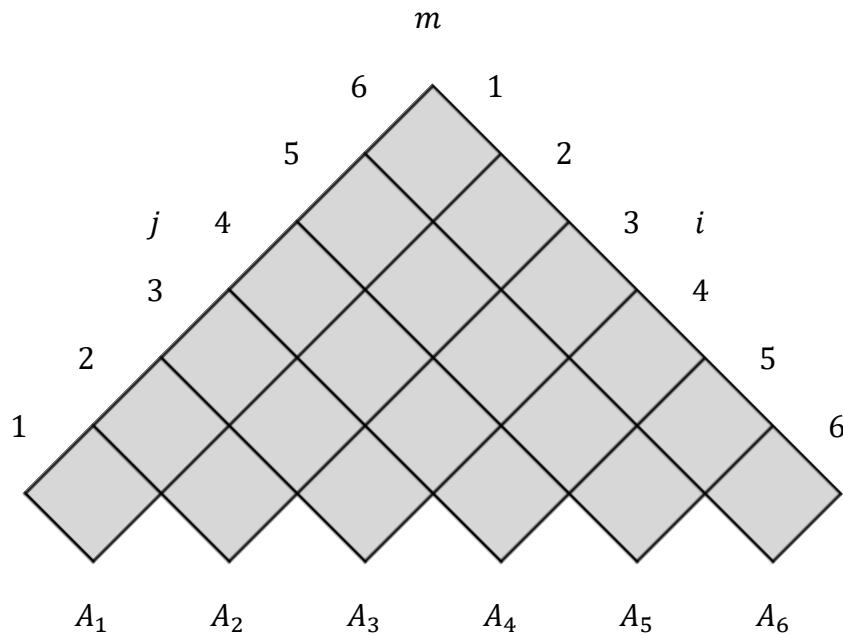
Matrix-Chain Mult: Bottom-up DP

MATRIX-CHAIN-ORDER (p)

1. $n \leftarrow p.length - 1$
2. $m[1..n, 1..n] \leftarrow \text{new table}$, $s[1..n-1, 2..n] \leftarrow \text{new table}$
3. **for** $i \leftarrow 1$ **to** n **do**
4. $m[i, i] \leftarrow 0$
5. **for** $l \leftarrow 2$ **to** n **do** *// l is the chain length*
6. **for** $i \leftarrow 1$ **to** $n-l+1$ **do**
7. $j \leftarrow i + l - 1$
8. $m[i, j] \leftarrow \infty$
9. **for** $k \leftarrow i$ **to** $j-1$ **do**
10. $q \leftarrow m[i, k] + m[k+1, j] + p_{i-1}p_kp_j$
11. **if** $q < m[i, j]$ **then**
12. $m[i, j] \leftarrow q$
13. $s[i, j] \leftarrow k$
14. **return** m **and** s

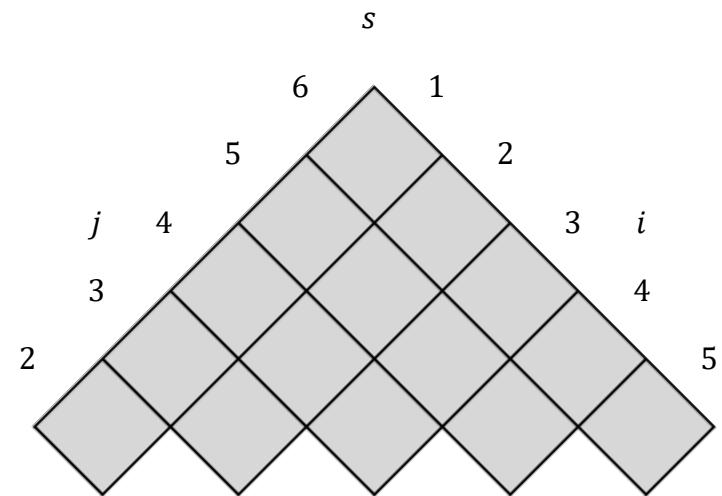
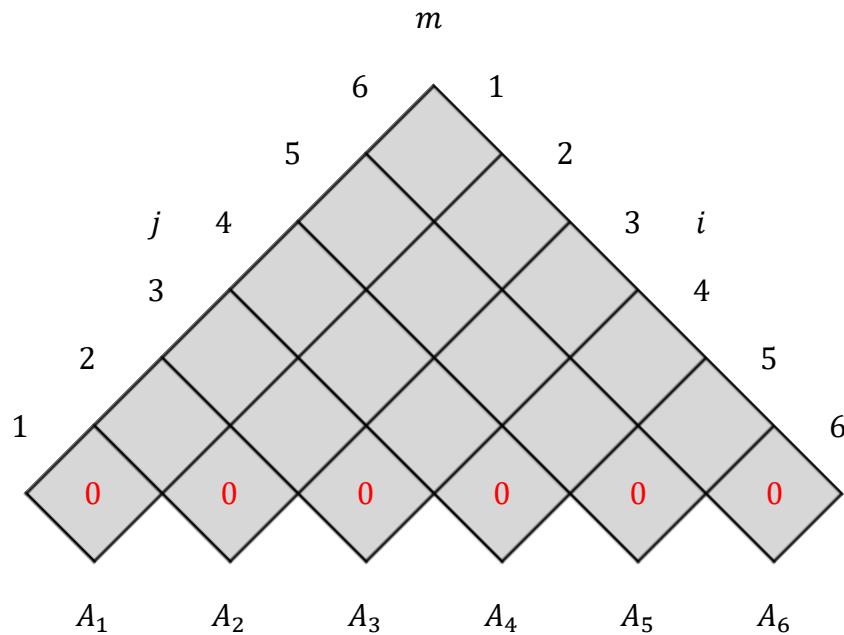
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 $(p_0 \times p_1)$	35×15 $(p_1 \times p_2)$	15×5 $(p_2 \times p_3)$	5×10 $(p_3 \times p_4)$	10×20 $(p_4 \times p_5)$	20×25 $(p_5 \times p_6)$



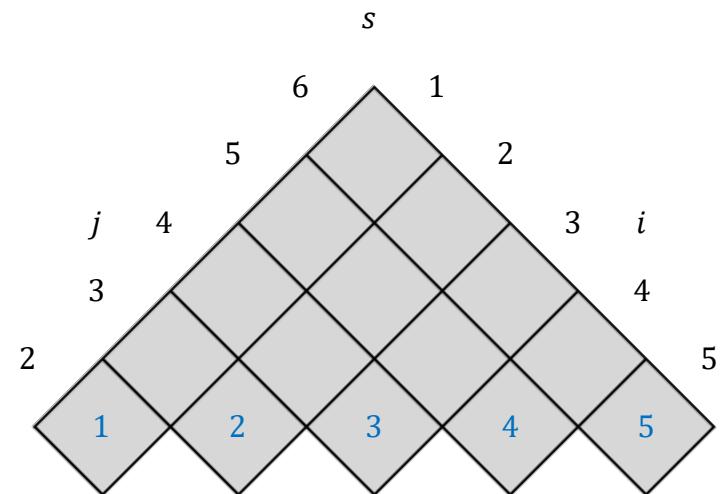
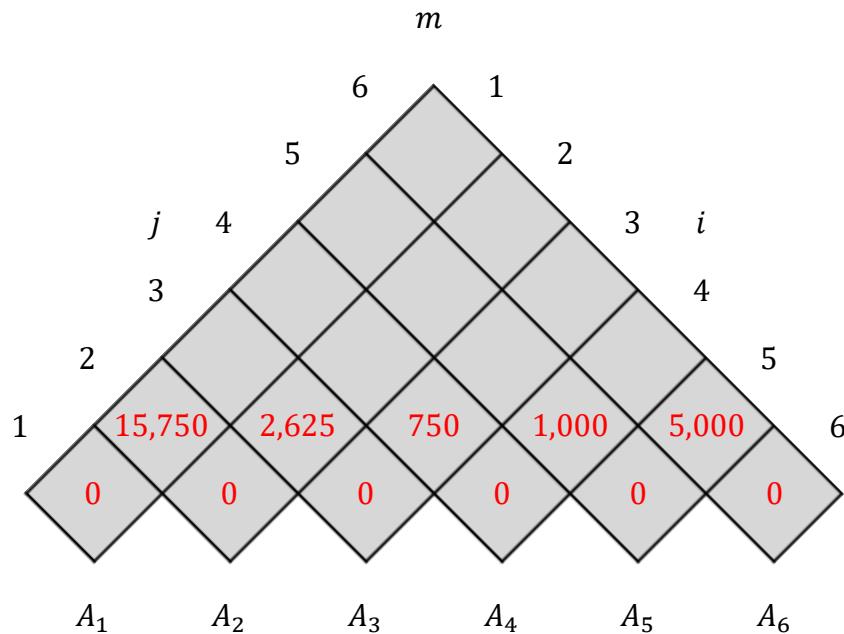
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 $(p_0 \times p_1)$	35×15 $(p_1 \times p_2)$	15×5 $(p_2 \times p_3)$	5×10 $(p_3 \times p_4)$	10×20 $(p_4 \times p_5)$	20×25 $(p_5 \times p_6)$



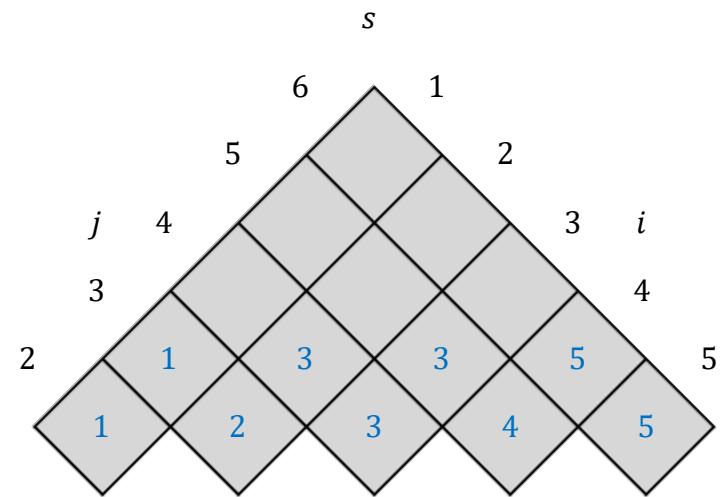
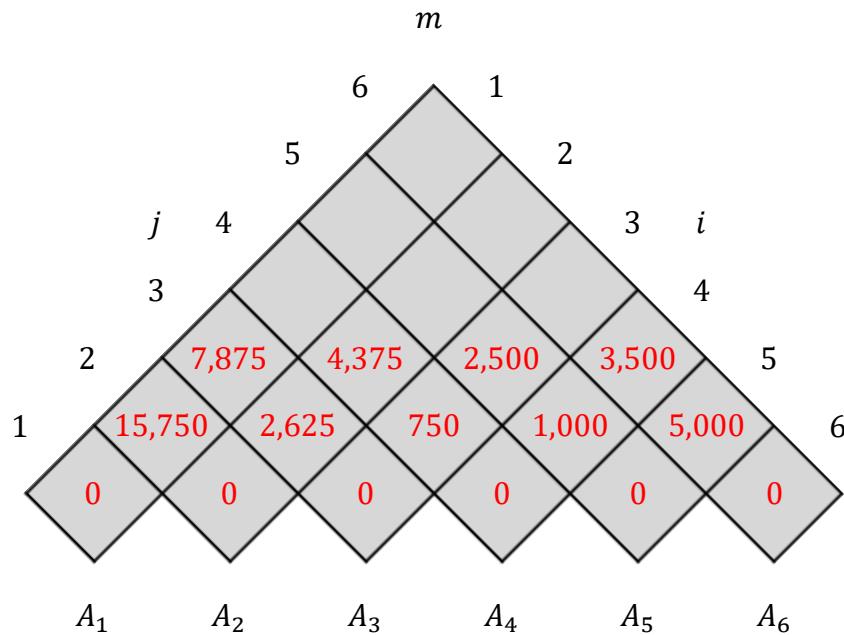
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 $(p_0 \times p_1)$	35×15 $(p_1 \times p_2)$	15×5 $(p_2 \times p_3)$	5×10 $(p_3 \times p_4)$	10×20 $(p_4 \times p_5)$	20×25 $(p_5 \times p_6)$



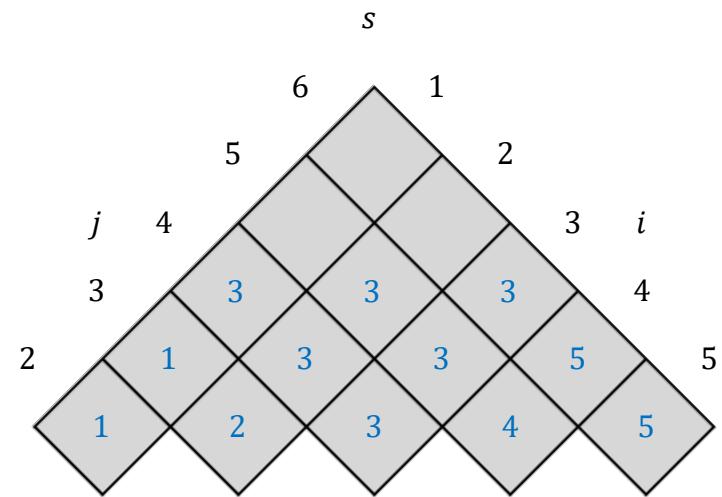
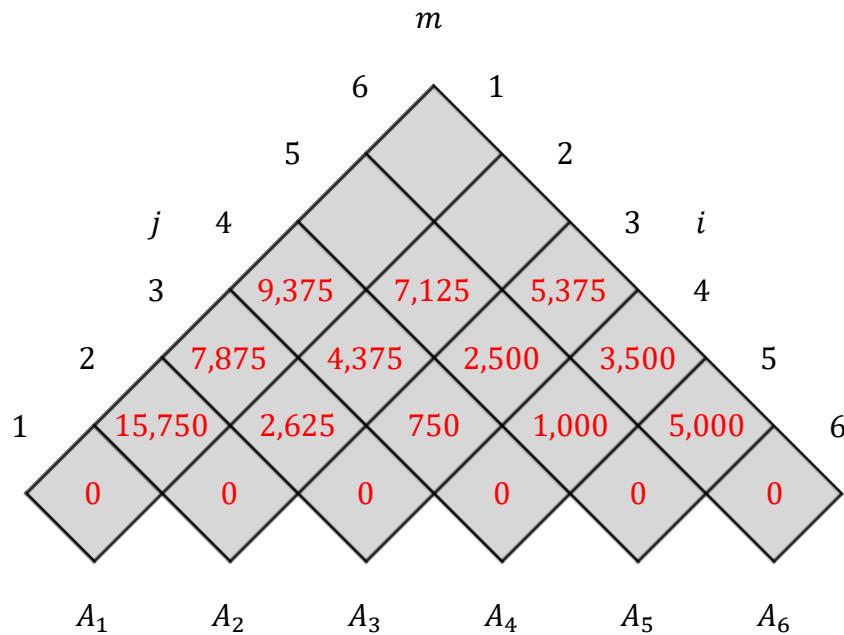
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 $(p_0 \times p_1)$	35×15 $(p_1 \times p_2)$	15×5 $(p_2 \times p_3)$	5×10 $(p_3 \times p_4)$	10×20 $(p_4 \times p_5)$	20×25 $(p_5 \times p_6)$



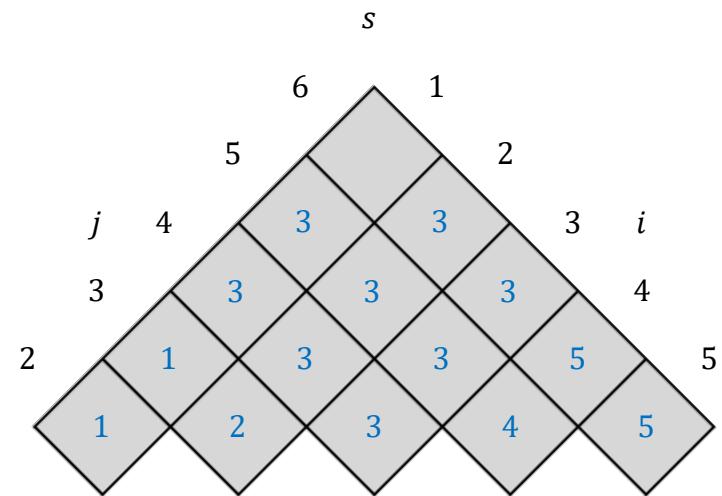
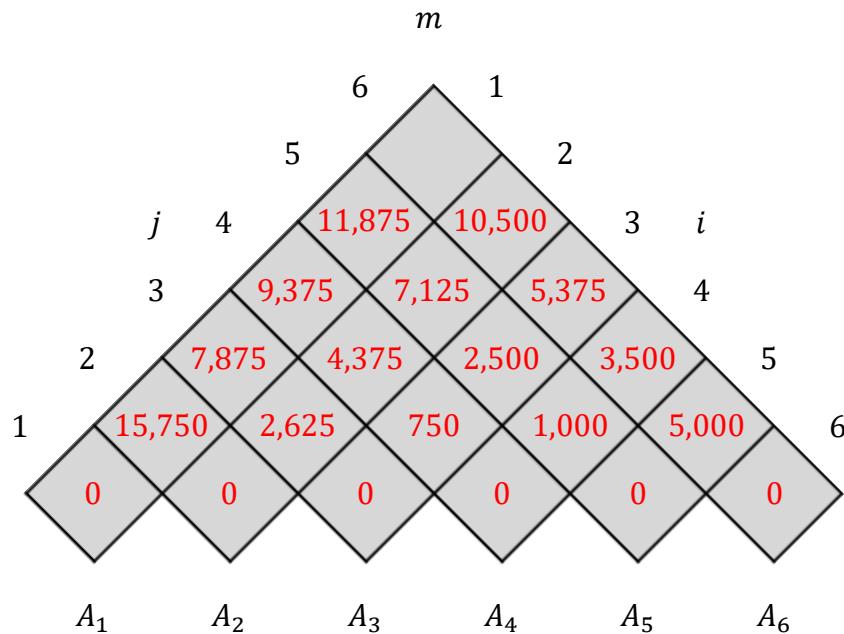
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



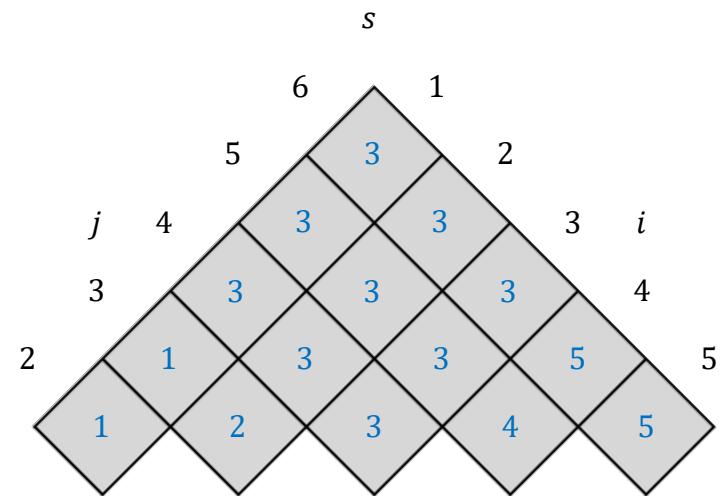
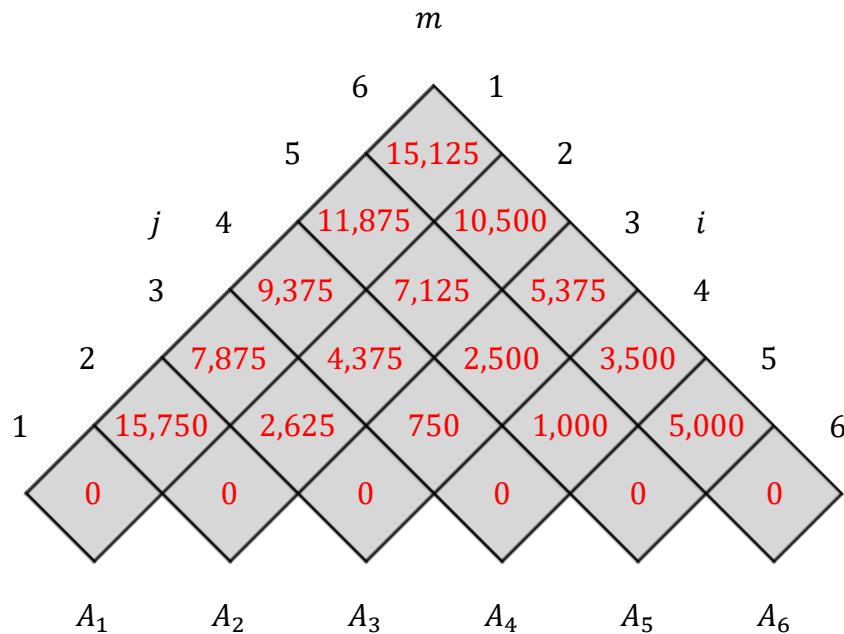
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



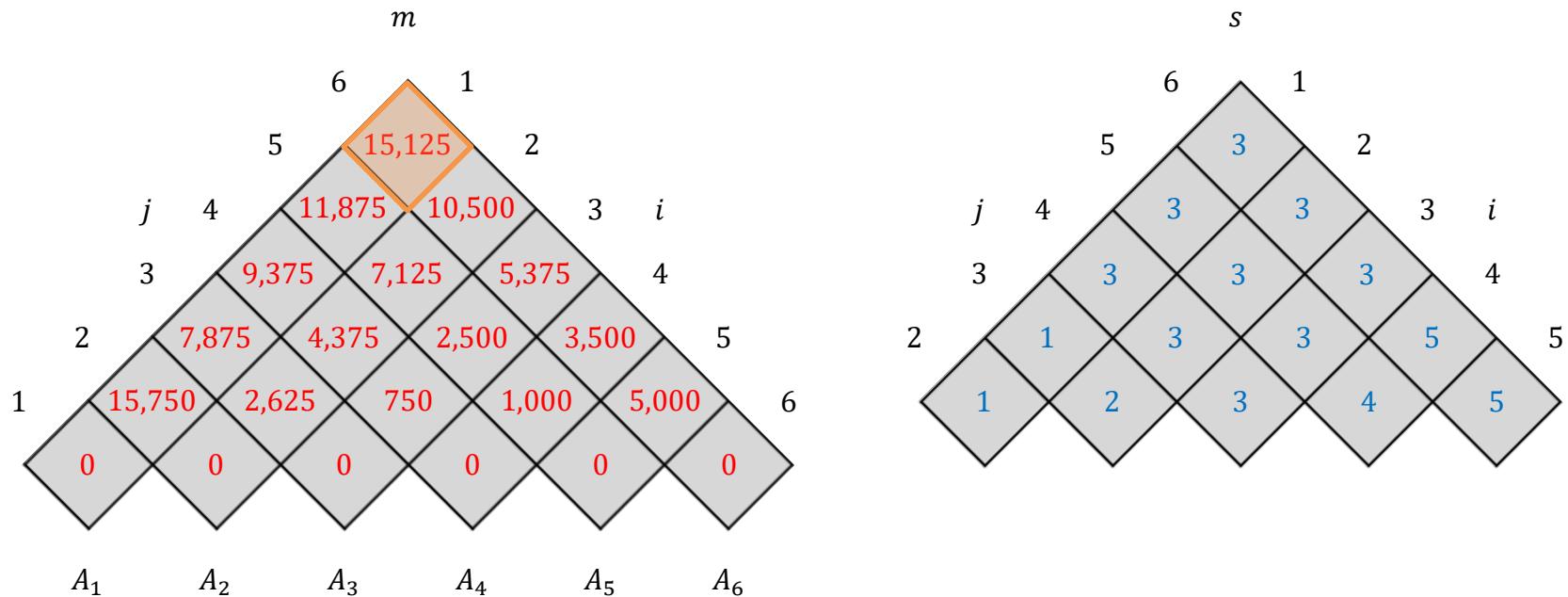
Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



Matrix-Chain Mult: Bottom-up DP

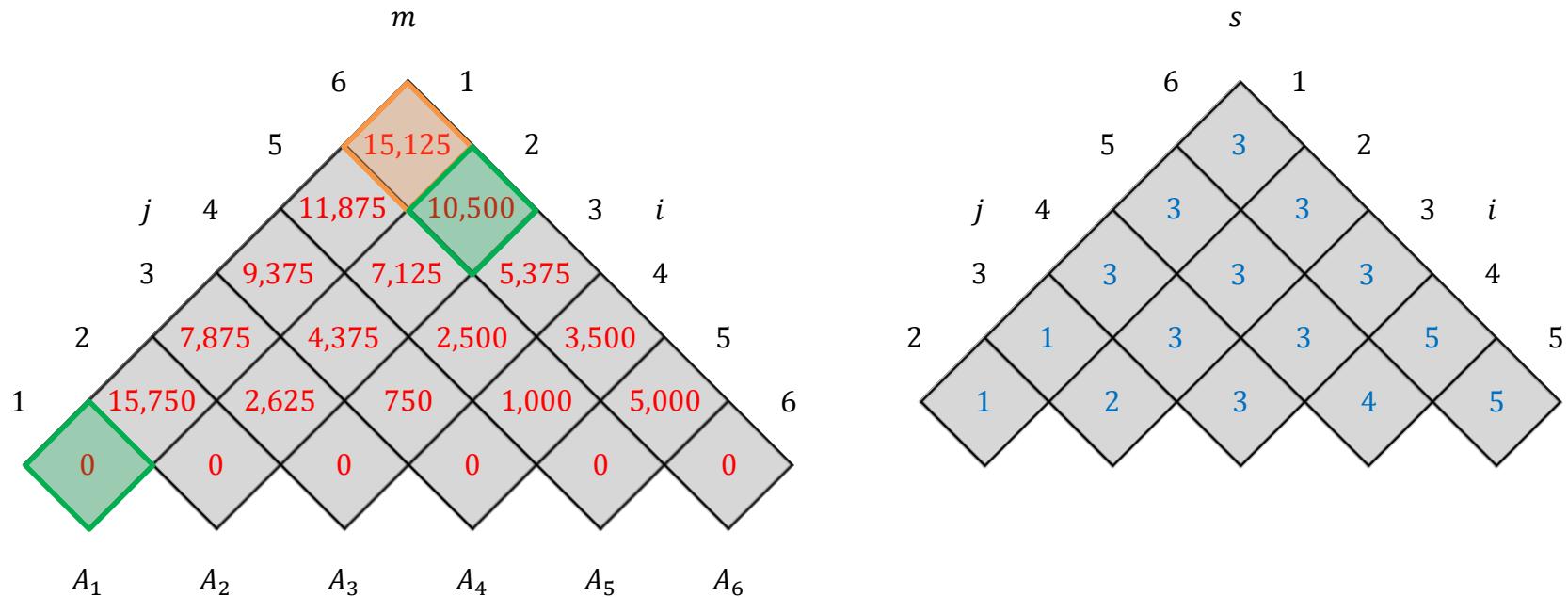
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1, 6]$ $\left(= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k < 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\} \right)$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)

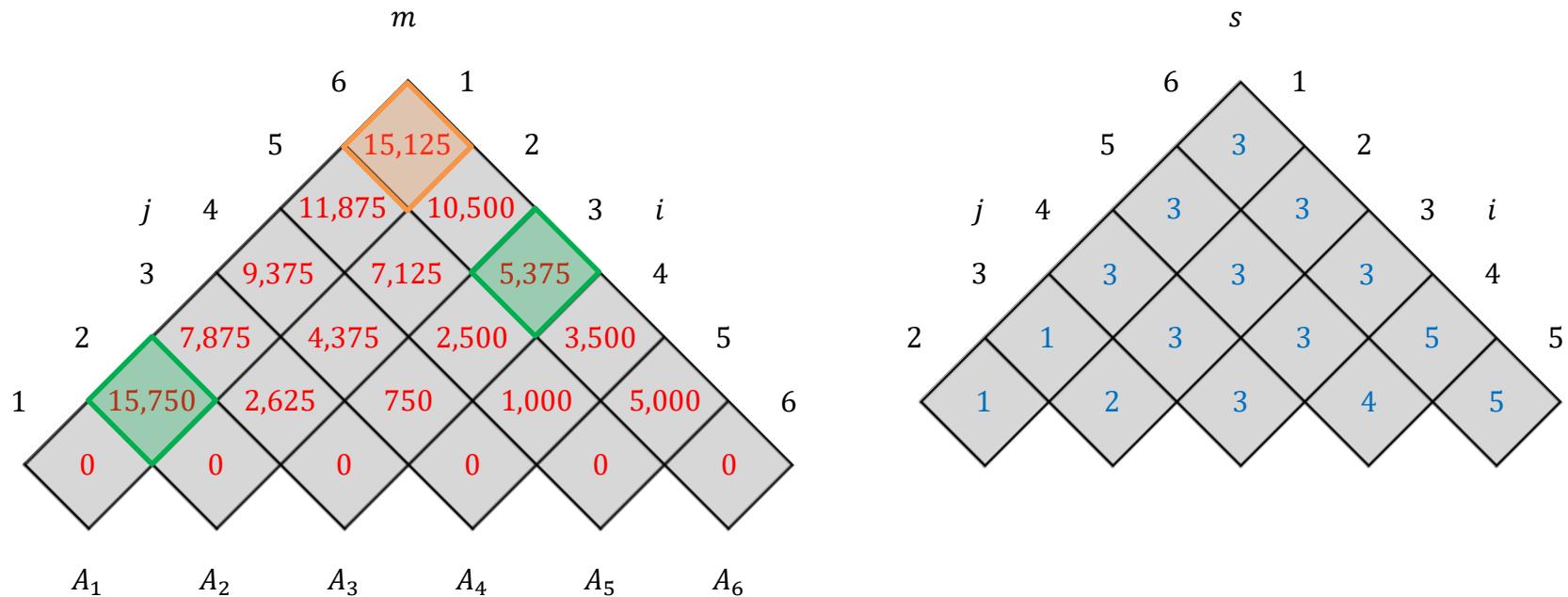


To compute $m[1, 6]$ $\left(= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\}\right)$

$$[k = 1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



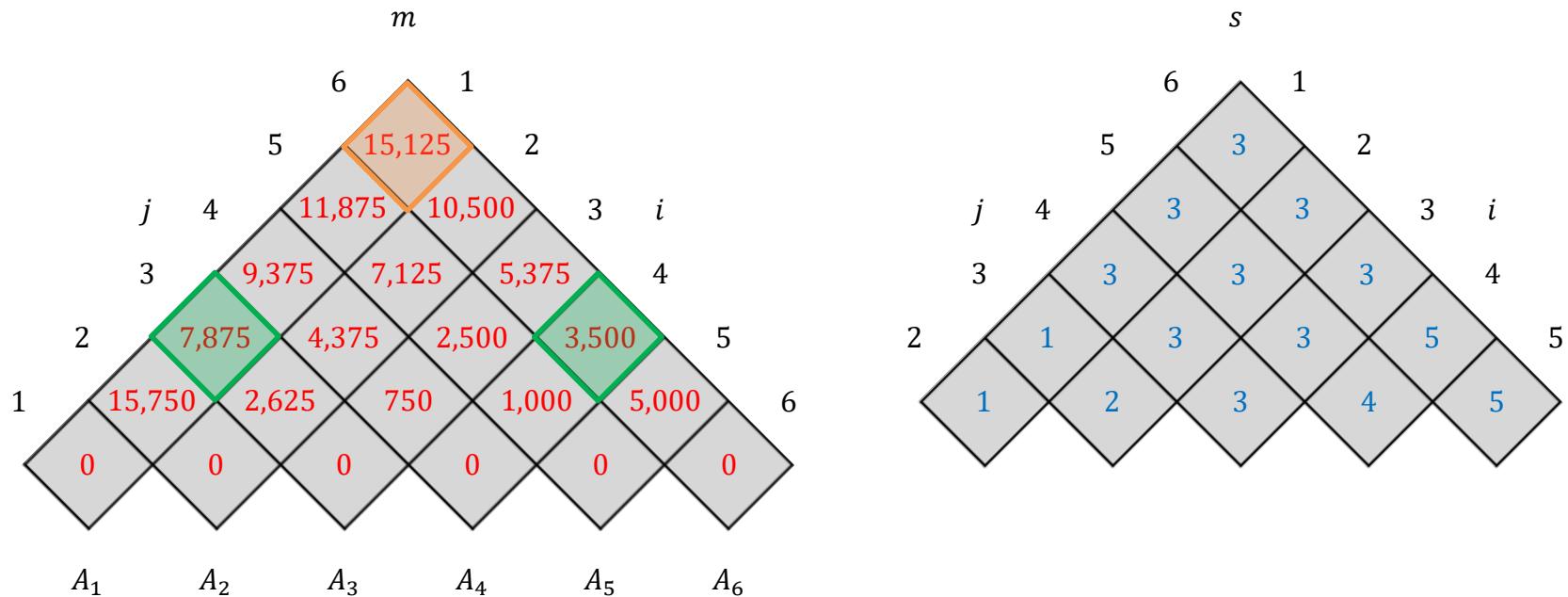
To compute $m[1, 6]$ ($= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\}$)

$$[k = 1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

$$[k = 2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1, 6]$ ($= \min_{i \leq k < j} \{m(i, k) + m(k+1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k+1, 6) + p_0p_kp_6\}$)

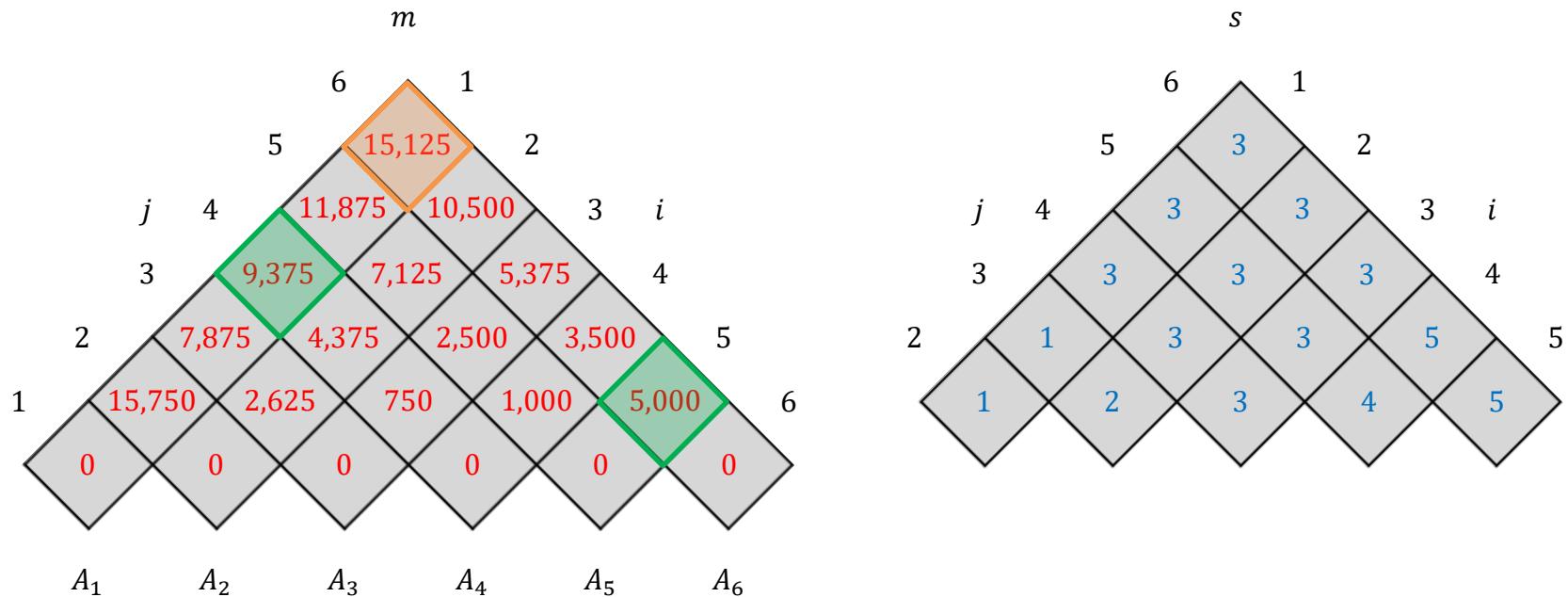
$$[k=1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

$$[k=2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375$$

$$[k=3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1, 6]$ $\left(= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\} \right)$

$$[k = 1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

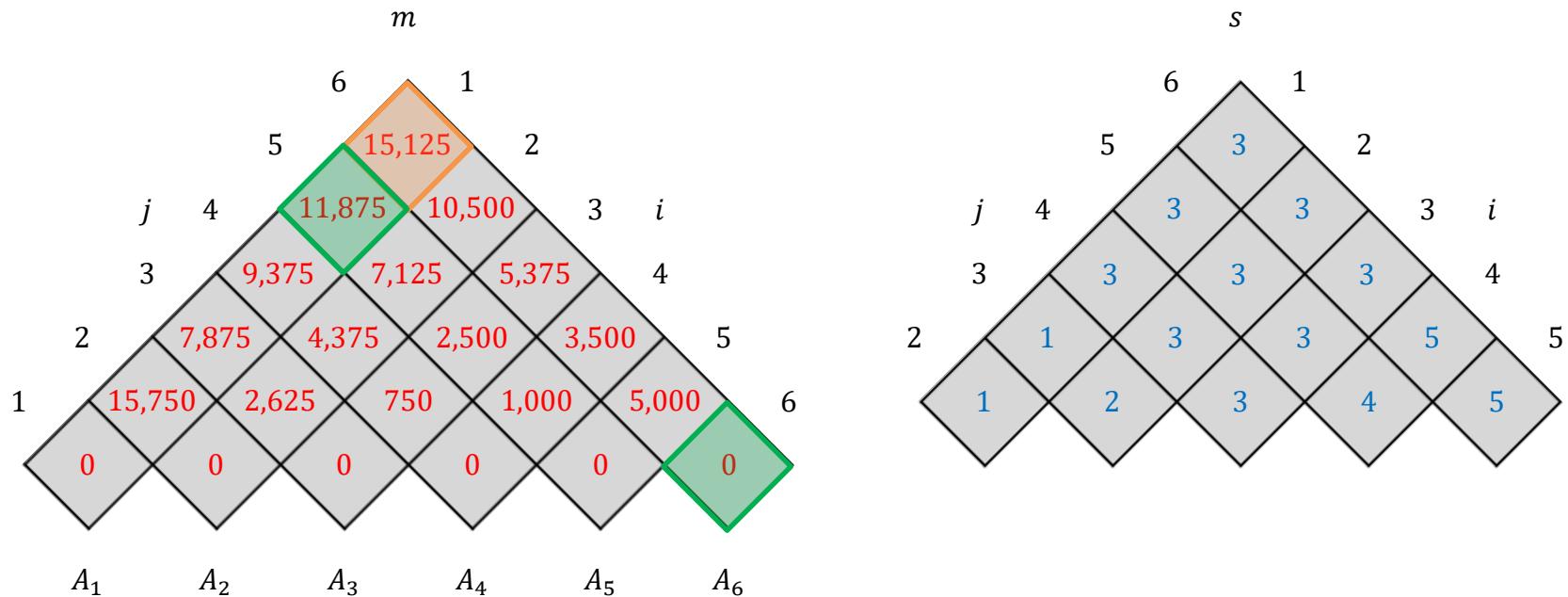
$$[k = 2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375$$

$$[k = 3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125$$

$$[k = 4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1, 6]$ ($= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\}$)

$$[k = 1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

$$[k = 2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375$$

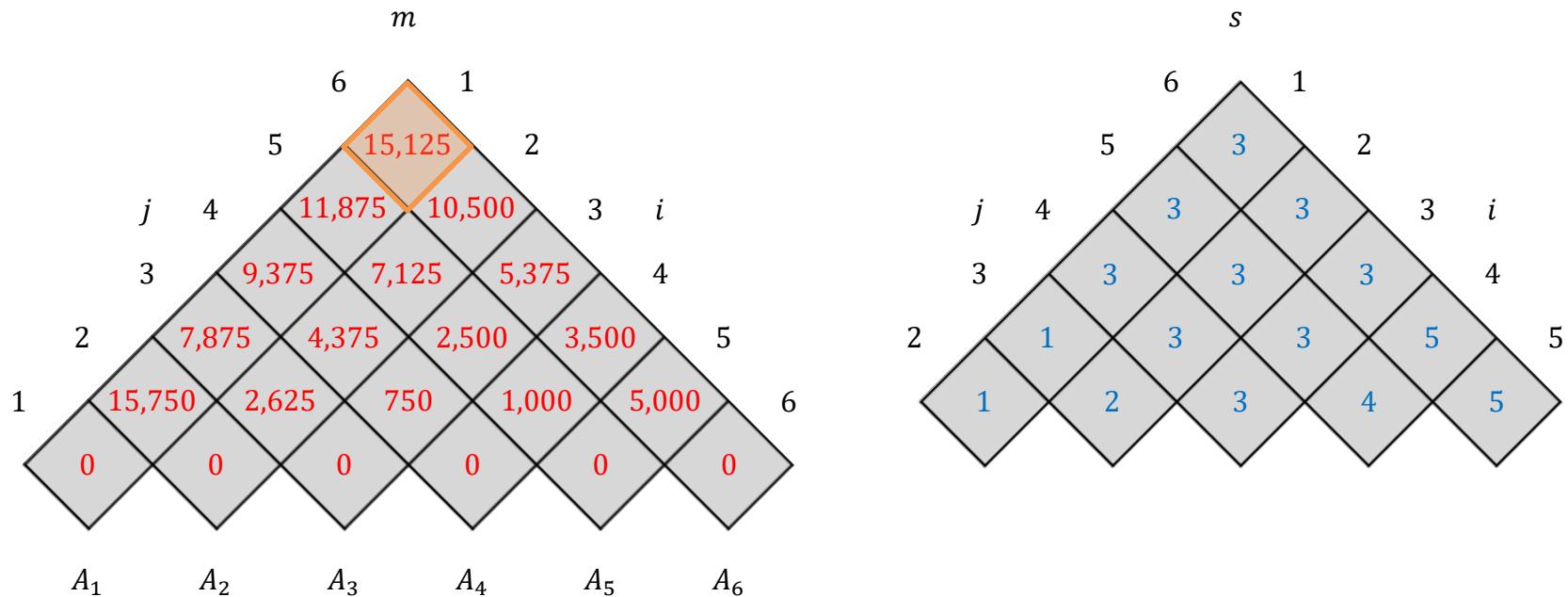
$$[k = 3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125$$

$$[k = 4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875$$

$$[k = 5] \Rightarrow m[1,5] + m[6,6] + p_0p_5p_6 = 11,875 + 0 + 30 \times 20 \times 25 = 26,875$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1,6]$ ($= \min_{i \leq k < j} \{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\} = \min_{1 \leq k < 6} \{m(1,k) + m(k+1,6) + p_0p_kp_6\}$)

$$[k=1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750$$

$$[k=2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375$$

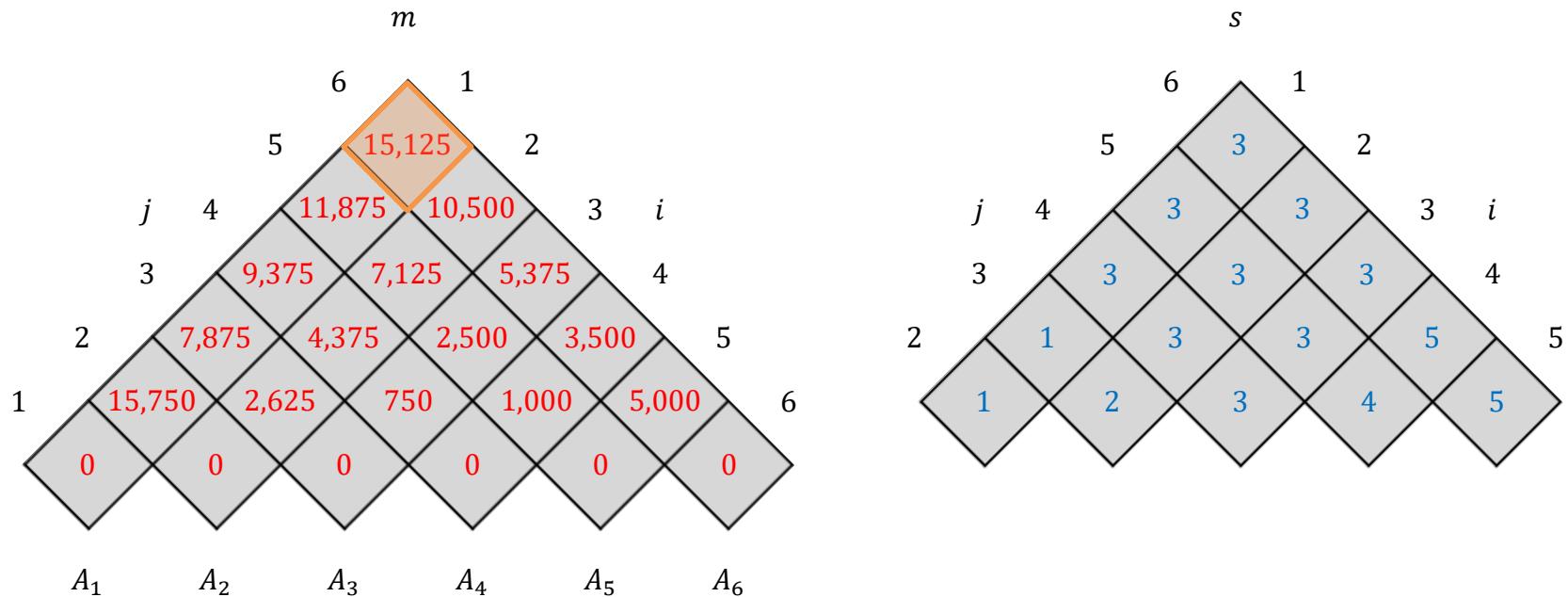
$$[k=3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125$$

$$[k=4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875$$

$$[k=5] \Rightarrow m[1,5] + m[6,6] + p_0p_5p_6 = 11,875 + 0 + 30 \times 20 \times 25 = 26,875$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)

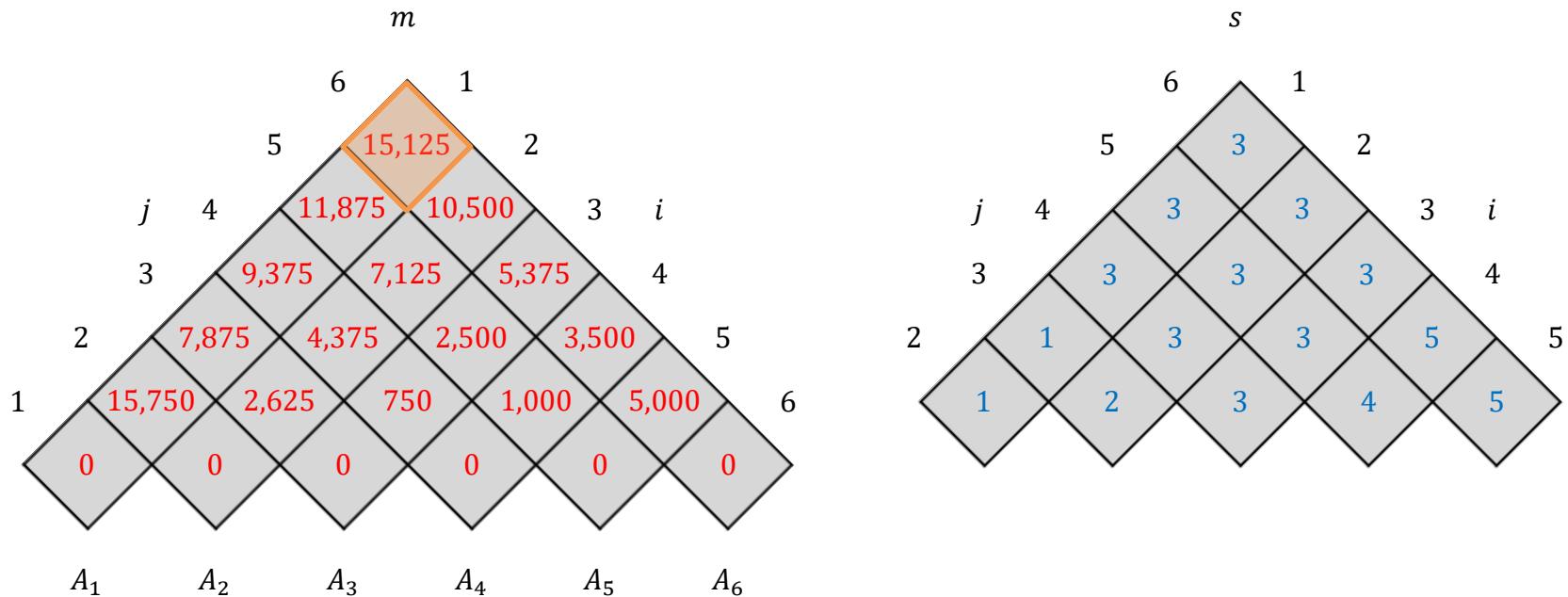


To compute $m[1, 6]$ ($= \min_{i \leq k < j} \{m(i, k) + m(k+1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k+1, 6) + p_0p_kp_6\}$)

$$\min \left[\begin{array}{l} [k=1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750 \\ [k=2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375 \\ [k=3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125 \\ [k=4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875 \\ [k=5] \Rightarrow m[1,5] + m[6,6] + p_0p_5p_6 = 11,875 + 0 + 30 \times 20 \times 25 = 26,875 \end{array} \right]$$

Matrix-Chain Mult: Bottom-up DP

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)

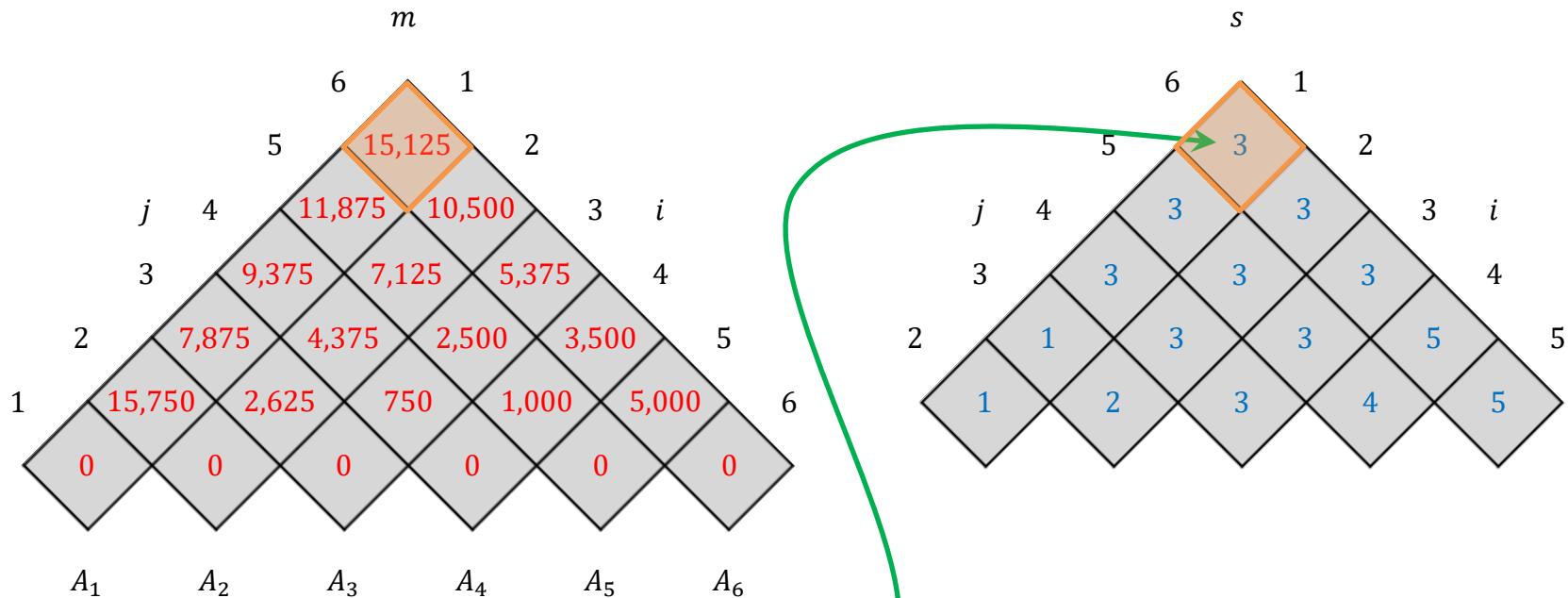


To compute $m[1, 6]$ ($= \min_{i \leq k < j} \{m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1, k) + m(k + 1, 6) + p_0p_kp_6\}$)

$$\min \left[\begin{array}{l} [k = 1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750 \\ [k = 2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375 \\ [k = 3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125 \\ [k = 4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875 \\ [k = 5] \Rightarrow m[1,5] + m[6,6] + p_0p_5p_6 = 11,875 + 0 + 30 \times 20 \times 25 = 26,875 \end{array} \right]$$

Matrix-Chain Mult: Bottom-up DP

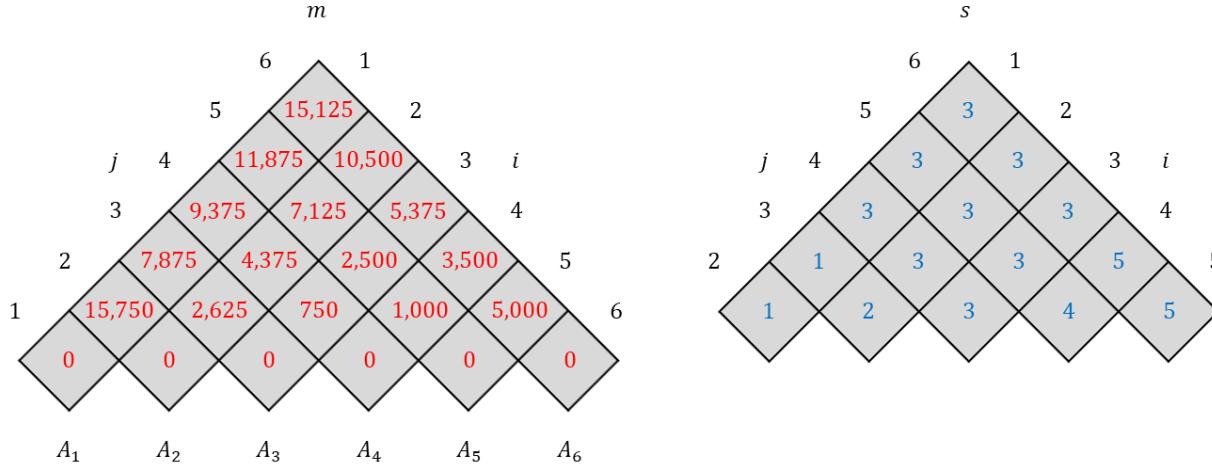
matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35 ($p_0 \times p_1$)	35×15 ($p_1 \times p_2$)	15×5 ($p_2 \times p_3$)	5×10 ($p_3 \times p_4$)	10×20 ($p_4 \times p_5$)	20×25 ($p_5 \times p_6$)



To compute $m[1,6]$ ($= \min_{i \leq k \leq j} \{m(i,k) + m(k+1,j) + p_{i-1}p_kp_j\} = \min_{1 \leq k \leq 6} \{m(1,k) + m(k+1,6) + p_0p_kp_6\}$)

- $\min \left[\begin{array}{l} [k=1] \Rightarrow m[1,1] + m[2,6] + p_0p_1p_6 = 0 + 10,500 + 30 \times 35 \times 25 = 36,750 \\ [k=2] \Rightarrow m[1,2] + m[3,6] + p_0p_2p_6 = 15,750 + 5,375 + 30 \times 15 \times 25 = 32,375 \\ [k=3] \Rightarrow m[1,3] + m[4,6] + p_0p_3p_6 = 7,875 + 3,500 + 30 \times 5 \times 25 = 15,125 \\ [k=4] \Rightarrow m[1,4] + m[5,6] + p_0p_4p_6 = 9,375 + 5,000 + 30 \times 10 \times 25 = 21,875 \\ [k=5] \Rightarrow m[1,5] + m[6,6] + p_0p_5p_6 = 11,875 + 0 + 30 \times 20 \times 25 = 26,875 \end{array} \right]$

Matrix-Chain Mult: Extracting the Solution



matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

*PRINT-OPTIMAL-PARENS (*s*, *i*, *j*)*

1. *if* $i = j$ *then*
2. *print* “ A_i ”
3. *else print* “(”
4. *PRINT-OPTIMAL-PARENS (*s*, *i*, *s*[*i*,*j*])*
5. *PRINT-OPTIMAL-PARENS (*s*, *s*[*i*,*j*] + 1, *j*)*
6. *print* “)”

Dynamic Programming vs. Divide-and-Conquer

- Dynamic programming, like the divide-and-conquer method, solves problems by combining solutions to subproblems
- Divide-and-conquer algorithms
 - partition the problem into disjoint subproblems,
 - solve the subproblems recursively, and
 - then combine their solutions to solve the original problem
- In contrast, dynamic programming applies when the subproblems overlap — that is, when subproblems share subsubproblems
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Elements of Dynamic Programming

An optimization problem must have the following two ingredients for dynamic programming to apply.

- 1) Optimal substructure
 - an optimal solution to the problem contains within it optimal solutions to subproblems
- 2) Overlapping subproblems
 - subproblems share subsubproblems and/or subsubsubproblems and/or subsubsubsubproblems, and so on

Dynamic Programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

- 1) Characterize the structure of an optimal solution.
- 2) Recursively define the value of an optimal solution.
- 3) Compute the value of an optimal solution, typically in a bottom-up fashion.
- 4) Construct an optimal solution from computed information.

If we need only the value of an optimal solution, and not the solution itself, then we can omit step 4.

If we perform step 4, we sometimes maintain additional information during step 3 so that we can easily construct an optimal solution.

Longest Common Subsequence (LCS)

A *subsequence* of a sequence X is obtained by deleting zero or more symbols from X .

Example:

$$X = abcba$$

$Z = bca \leftarrow$ obtained by deleting the 1st ' a ' and the 2nd ' b ' from X

A *Longest Common Subsequence (LCS)* of two sequences X and Y is a sequence Z that is a subsequence of both X and Y , and is the longest among all such subsequences.

Given X and Y , the *LCS problem* asks for such a Z .

LCS: Optimal Substructure

Given two sequences: $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$

Let $Z = \langle z_1, z_2, \dots, z_k \rangle$ be any LCS of X and Y .

For $0 \leq i \leq m$, let $X_i = \langle x_1, x_2, \dots, x_i \rangle$. We define Y_i and Z_i similarly.

Then

(1) If $x_m = y_n$,

then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1} .

(2) If $x_m \neq y_n$,

then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y .

(3) If $x_m \neq y_n$,

then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1} .

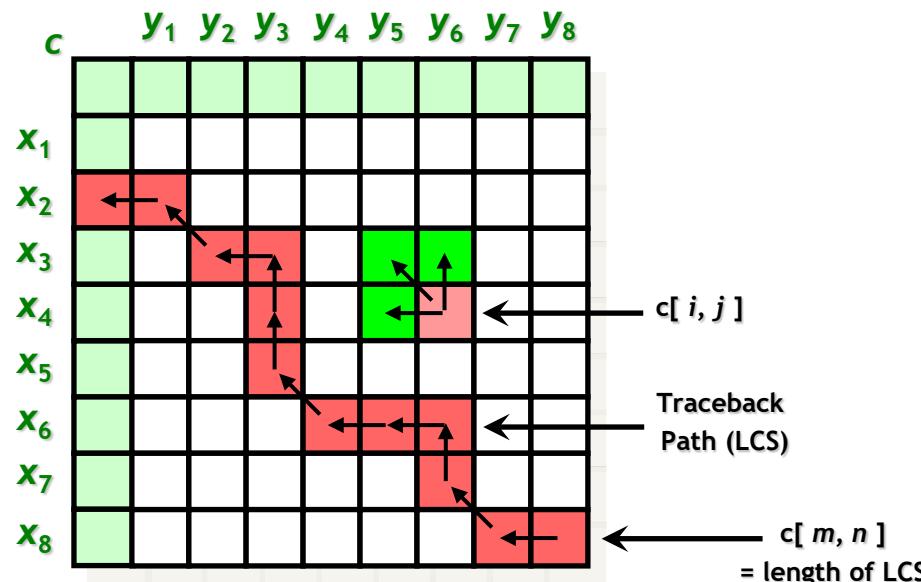
LCS: Recurrence

Given two sequences: $X = \langle x_1, x_2, \dots, x_m \rangle$ and $Y = \langle y_1, y_2, \dots, y_n \rangle$

For $0 \leq i \leq m$ and $0 \leq j \leq n$,

let $c[i, j]$ be the length of an LCS of X_i and Y_j . Then

$$c[i, j] = \begin{cases} 0, & \text{if } i = 0 \vee j = 0, \\ c[i - 1, j - 1] + 1, & \text{if } i, j > 0 \wedge x_i = y_j, \\ \max\{c[i, j - 1], c[i - 1, j]\}, & \text{otherwise.} \end{cases}$$



LCS: Bottom-up DP

LCS-LENGTH (X, Y)

1. $m \leftarrow X.length$
2. $n \leftarrow Y.length$
3. $b[1 \dots m, 1 \dots n] \leftarrow \text{new table}$, $c[0 \dots m, 0 \dots n] \leftarrow \text{new table}$
4. **for** $i \leftarrow 1$ **to** m
5. $c[i, 0] \leftarrow 0$
6. **for** $j \leftarrow 0$ **to** n
7. $c[0, j] \leftarrow 0$
8. **for** $i \leftarrow 1$ **to** m
9. **for** $j \leftarrow 1$ **to** n
10. **if** $x_i = y_j$
11. $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
12. $b[i, j] \leftarrow \text{“↖”}$
13. **elseif** $c[i - 1, j] \geq c[i, j - 1]$
14. $c[i, j] \leftarrow c[i - 1, j]$
15. $b[i, j] \leftarrow \text{“↑”}$
16. **else** $c[i, j] \leftarrow c[i, j - 1]$
17. $b[i, j] \leftarrow \text{“←”}$

Running time = $\Theta(mn)$

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i							
0							
1	A						
2	B						
3	C						
4	B						
5	D						
6	A						
7	B						

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0					
2	B	0					
3	C	0					
4	B	0					
5	D	0					
6	A	0					
7	B	0					

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0					
3	C	0					
4	B	0					
5	D	0					
6	A	0					
7	B	0					

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0					
4	B	0					
5	D	0					
6	A	0					
7	B	0					

Annotations in the table:

- Green arrows (\uparrow) indicate matches between x_i and y_j .
- Red arrows (\leftarrow) indicate matches between x_i and y_j , where y_j is part of a local subsequence.
- Blue numbers ($1, 2$) indicate matches between x_i and y_j , where y_j is part of a global subsequence.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	↖	2	2
4	B	0					
5	D	0					
6	A	0					
7	B	0					

Annotations in the table:

- Green arrows point up from row 3 to row 2.
- Red arrows point left from row 2 to row 1.
- Blue arrows point left from row 3 to row 2.
- Red numbers 1, 2, and 3 are placed in cells (1,2), (2,3), and (3,4) respectively.
- Blue numbers 1, 2, and 2 are placed in cells (2,3), (3,4), and (3,5) respectively.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	← 2	2
4	B	0	1	1	2	2	3
5	D	0					
6	A	0					
7	B	0					

Annotations in the table:

- Green arrows point up from row 0 to row 1.
- Red arrows point left from row 1 to row 2.
- Blue arrows point left from row 2 to row 3.
- Red numbers (1, 2, 3) are placed in cells where a red arrow points left.
- Green numbers (0, 1, 2, 3) are placed in cells where a green arrow points up.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0					
7	B	0					

The table illustrates the bottom-up dynamic programming solution for the Longest Common Subsequence (LCS) problem. The rows represent the sequence x_i and the columns represent the sequence y_j . The values in the cells indicate the length of the LCS up to that point. Colored arrows show the path of decisions:

- Green Upward Arrows:** Indicate matches where $x_i = y_j$.
- Blue Leftward Arrows:** Indicate matches where $x_i \neq y_j$ and the previous character in x_i is included in the subsequence.
- Red Diagonal Arrows:** Indicate matches where $x_i \neq y_j$ and the previous character in y_j is included in the subsequence.

The final value in the bottom-right cell is 3, indicating the length of the LCS between the two sequences.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0					

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	← 2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

The table illustrates the bottom-up dynamic programming approach for finding the Longest Common Subsequence (LCS) between two sequences, x_i and y_j . The rows represent the sequence x_i and the columns represent the sequence y_j . The cells contain the length of the LCS up to that point, with arrows indicating the direction of the LCS path:

- Green Upward Arrows:** Indicate matches where x_i and y_j have the same character.
- Red Leftward Arrows:** Indicate matches where x_i and y_j have different characters, and the character in x_i is part of the LCS.
- Cyan Leftward Arrows:** Indicate matches where x_i and y_j have different characters, and the character in y_j is part of the LCS.

The final value in the bottom-right cell is 4, indicating the length of the LCS between the two sequences. The character 'A' at index 6 is circled in black, and the value 4 at index 6 is circled in red, highlighting the result.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	A	0	1	2	2	3	4
7	B	0	1	2	2	3	4

The table illustrates the bottom-up dynamic programming approach for finding the Longest Common Subsequence (LCS) between two sequences, x and y . The rows represent the sequence y (labeled i) and the columns represent the sequence x (labeled j). The entries in the table are the lengths of the LCS up to that point. Colored arrows indicate the operations used to build the LCS:

- Green Upward Arrows:** Indicate matches where $x_j = y_i$.
- Red Leftward Arrows:** Indicate matches where $x_j = y_{i-1}$.
- Cyan Leftward Arrows:** Indicate matches where $x_j = y_{i-2}$.

The final value in the bottom-right cell (A at $j=6$, B at $i=7$) is 4, which is the length of the LCS. The path of arrows shows the sequence of operations used to reach this result.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	(B)	(A)
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	2	2
4	(B)	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	(A)	0	1	2	2	3	4
7	B	0	1	2	2	3	4

The table illustrates the bottom-up Dynamic Programming (DP) approach for finding the Longest Common Subsequence (LCS) between two sequences, A and B. The rows represent sequence A (y_j) and the columns represent sequence B (x_i). The values in the cells indicate the length of the LCS up to that point. Colored arrows show the path of decisions:

- Green Upward Arrows:** Indicate matches where both characters are the same.
- Red Leftward Arrows:** Indicate matches where the character in A is different, so the path moves left.
- Cyan Leftward Arrows:** Indicate matches where the character in B is different, so the path moves left.

Cells containing circled numbers (3 and 4) highlight specific points of interest in the sequence B column. The final length of the LCS is 4, which is also indicated by the circled value in the last cell of the sequence B column.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	C	A	(B)	(A)
0	x_i	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	C	0	1	1	2	← 2	2
4	(B)	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	(A)	0	1	2	2	3	4
7	B	0	1	2	2	3	4

The table illustrates the bottom-up dynamic programming solution for the Longest Common Subsequence (LCS) problem. The rows represent the sequence x_i and the columns represent the sequence y_j . The values in the cells indicate the length of the LCS up to that point. Colored arrows (green up, red left, blue left) show the local moves used to build the sequence. The final lengths are circled in black.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	\textcircled{C}	A	\textcircled{B}	\textcircled{A}
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	← 1	← 1	1	2
3	\textcircled{C}	0	1	1	↖	2	2
4	\textcircled{B}	0	1	1	2	2	3
5	D	0	1	2	2	2	3
6	\textcircled{A}	0	1	2	2	3	4
7	B	0	1	2	2	3	4

Annotations in the table:

- Green arrows point up or left, indicating matches or matches followed by insertions.
- Red arrows point right or down, indicating matches followed by deletions.
- Blue arrows point left, indicating matches followed by insertions.
- Red numbers (2, 3, 4) are circled in red and highlighted with a gray background, likely marking specific states or steps in the algorithm.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	B	D	\textcircled{C}	A	\textcircled{B}	\textcircled{A}
x_i	0	0	0	0	0	0	0
1 A	0	0	0	0	1	1	1
2 B	0	1	1	1	1	2	2
3 \textcircled{C}	0	1	1	2	2	2	2
4 \textcircled{B}	0	1	1	2	2	3	3
5 D	0	1	2	2	2	3	3
6 \textcircled{A}	0	1	2	2	3	3	4
7 B	0	1	2	2	3	4	4

Annotations in the table:

- Green arrows point up or left, indicating matches or matches followed by insertions.
- Red arrows point right or down, indicating matches followed by deletions.
- Blue arrows point left, indicating matches followed by insertions.
- Cells containing circled numbers (2, 3, 4) are highlighted with a gray background.
- Cells containing circled letters (B, A) are also highlighted with a gray background.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	(B)	D	(C)	A	(B)	(A)
x_i	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	(B)	0	1	1	1	2	2
3	(C)	0	1	1	2	2	2
4	(B)	0	1	1	2	2	3
5	D	0	1	2	2	3	3
6	(A)	0	1	2	2	3	4
7	B	0	1	2	2	3	4

Annotations in the table:

- Green arrows point up or left, indicating matches or matches followed by insertions.
- Red arrows point right or down, indicating matches followed by deletions.
- Blue arrows point left, indicating matches followed by insertions.
- Red numbers (1, 2, 3, 4) are circled in red and placed at specific intersections to indicate steps in the dynamic programming process.
- Grey shaded regions highlight specific subproblems or states in the DP table.

LCS: Bottom-up DP

j	0	1	2	3	4	5	6
i	y_j	(B)	D	(C)	A	(B)	(A)
x_i	0	0	0	0	0	0	0
1 A	0	0	0	0	1	1	1
2 (B)	0	1	1	1	1	2	2
3 (C)	0	1	1	2	2	2	2
4 (B)	0	1	1	2	2	3	3
5 D	0	1	2	2	2	3	3
6 (A)	0	1	2	2	3	3	4
7 B	0	1	2	2	3	4	4

Annotations in the table:

- Green arrows point up or left, indicating matches or matches followed by insertions.
- Red arrows point right or down, indicating matches followed by deletions.
- Blue arrows point left, indicating matches followed by insertions.
- Red numbers (1, 2, 3, 4) are circled in red and placed at specific intersections to indicate the path of the longest common subsequence.

LCS: Constructing an LCS

PRINT-LCS (b, X, i, j)

1. *if* $i = 0$ *or* $j = 0$
2. *return*
3. *if* $b[i,j] = “↖”$
4. PRINT-LCS ($b, X, i - 1, j - 1$)
5. print x_i
6. *elseif* $b[i,j] = “↑”$
7. PRINT-LCS ($b, X, i - 1, j$)
8. *else* PRINT-LCS ($b, X, i, j - 1$)

Running time = $O(m + n)$

Longest Increasing Subsequence (LIS)

An *Increasing Subsequence* L of a given sequence $A = \langle a_1, a_2, \dots, a_n \rangle$ of numbers is obtained by deleting zero or more numbers from A such that every number $x \in L$ is larger than the number immediately preceding x in L .

A *Longest Increasing Subsequence (LIS)* of A has the maximum length among all increasing subsequences of A .

Longest Increasing Subsequence (LIS)

Let's augment the given sequence $A = \langle a_1, a_2, \dots, a_n \rangle$ to include a sentinel value $a_0 = -\infty$. Thus $\langle a_0, a_1, a_2, \dots, a_n \rangle$ is our augmented sequence.

Let $LIS(i)$ be the length of the longest increasing subsequence of $\langle a_i, a_{i+1}, \dots, a_n \rangle$ that starts at a_i .

Then

$$LIS(i) = 1 + \max_{i < j \leq n} \{ LIS(j) \mid a_j > a_i \}$$

Running time = $\Theta(n^2)$.

Subset Sum

Given an array $A[1..n]$ of n positive integers and a target integer T , determine if any subset of the numbers in A sum up to T .

Subset Sum

Given an array $A[1..n]$ of n positive integers and a target integer T , determine if any subset of the numbers in A sum up to T .

Let $S(i, t)$ be *True* iff some subset of $A[i..n]$ adds up to t .

Then

$$S(i, t) = \begin{cases} \text{True}, & \text{if } t = 0, \\ \text{False}, & \text{if } t < 0 \text{ or } i > n, \\ S(i + 1, t) \vee S(i + 1, t - A[i]), & \text{otherwise.} \end{cases}$$

Running time = $\Theta(nT)$.

The resulting DP algorithm is called a *pseudo-polynomial time algorithm* because its running time depends on the numeric value of the input.

The Knapsack Problem

You have a knapsack of integer weight capacity W .

There are n items to pick from with the i^{th} item having weight w_i and value v_i , where $1 \leq i \leq n$. All weight values are integers.

You need to pick the most valuable combination of items that fit in your knapsack

Unbounded Knapsack:

Pick as many copies of each item as you want.

0/1 Knapsack:

Pick at most one copy of each item.

The Knapsack Problem

You have a knapsack of integer weight capacity W .

There are n items to pick from with the i^{th} item having weight w_i and value v_i , where $1 \leq i \leq n$. All weight values are integers.

You need to pick the most valuable combination of items that fit in your knapsack

Unbounded Knapsack:

Pick as many copies of each item as you want.

Let $K(w)$ = maximum value achievable with a knapsack of capacity w .

Then $K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$

Running time = $\Theta(nW)$.

The Knapsack Problem

You have a knapsack of integer weight capacity W .

There are n items to pick from with the i^{th} item having weight w_i and value v_i , where $1 \leq i \leq n$. All weight values are integers.

You need to pick the most valuable combination of items that fit in your knapsack

0/1 Knapsack:

Pick at most one copy of each item.

Let $K(w, i) =$ maximum value achievable with a knapsack of capacity w and items $1, 2, \dots, i$.

Then $K(w, i) = \max\{K(w - w_i, i - 1) + v_i, K(w, i - 1)\}$

Running time = $\Theta(nW)$.

Optional Optimal Binary Search Trees

Optimal Binary Search Trees (OPBST)

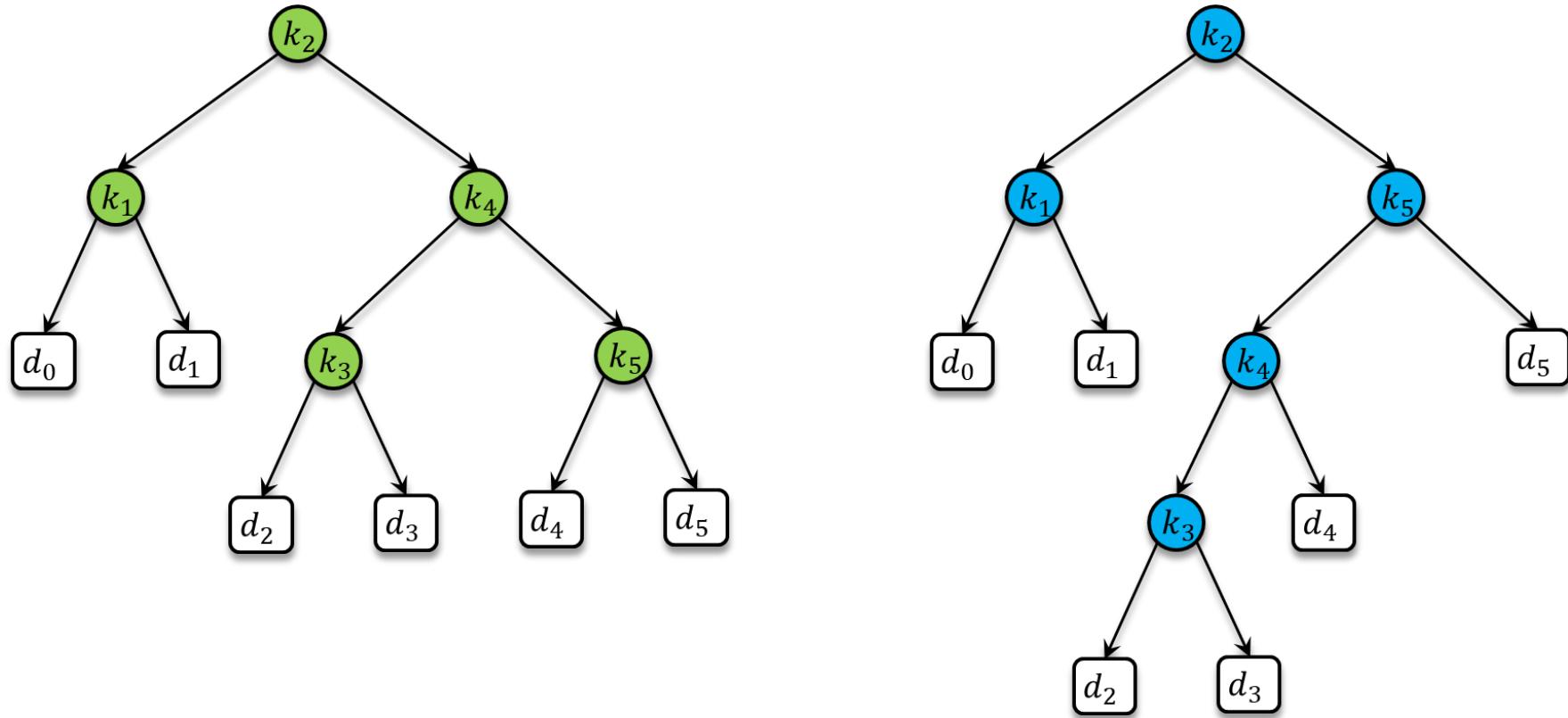
- Given (1) a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \dots < k_n$),
- (2) for $i \in [1, n]$, probability p_i that a search will be for k_i ,
 - (3) for $i \in [1, n - 1]$, probability q_i that a search will be for a key (say, d_i) between k_i and k_{i+1} ,
 - (4) probability q_0 that a search will be for a key (say, d_0) smaller than k_1 , and
 - (5) probability q_n that a search will be for a key (say, d_n) larger than k_n .

So, $\sum_{i=1}^n p_i + \sum_{i=0}^n q_i = 1$

Construct a binary search tree T from keys in K such that the following expected search cost in T is minimized:

$$\sum_{i=1}^n (\text{depth}(k_i) + 1). p_i + \sum_{i=0}^n (\text{depth}(d_i) + 1). q_i$$

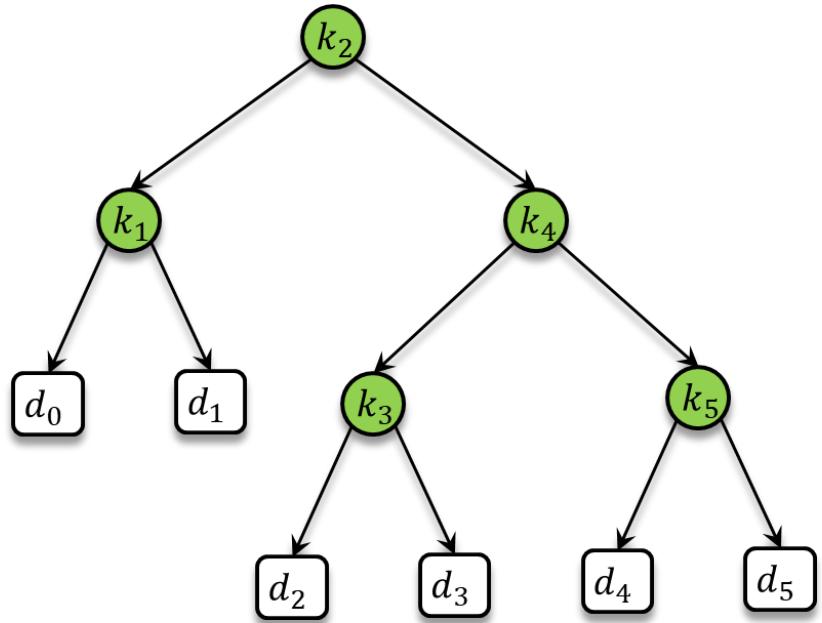
Optimal Binary Search Trees (OPBST)



k_i	k_1	k_2	k_3	k_4	k_5
p_i	0.15	0.10	0.05	0.10	0.20

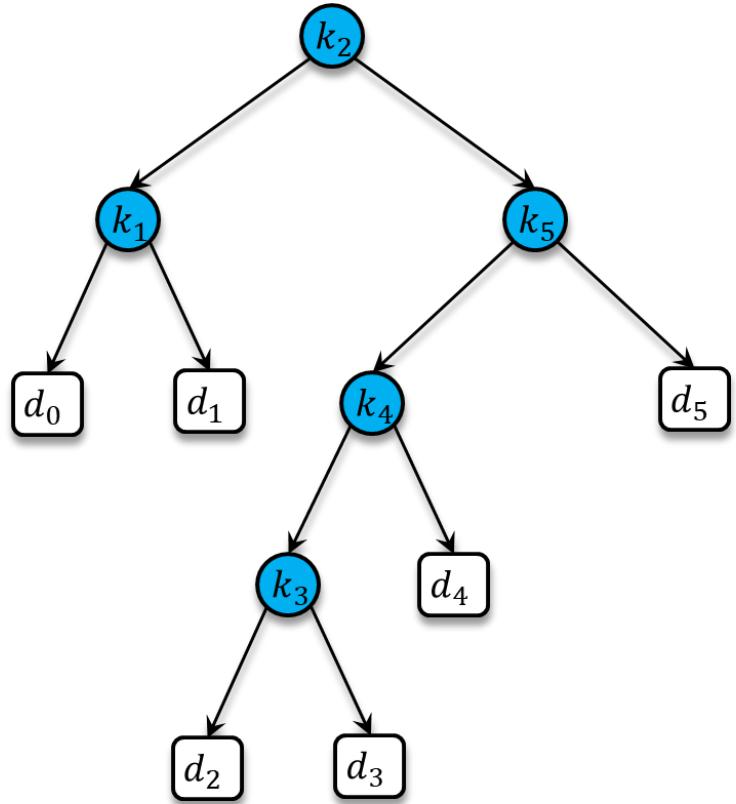
d_i	d_0	d_1	d_2	d_3	d_4	d_5
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Optimal Binary Search Trees (OPBST)



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Binary Search Trees (OPBST)



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	3	0.05	0.20
k_4	2	0.10	0.30
k_5	1	0.20	0.40
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	4	0.05	0.25
d_3	4	0.05	0.25
d_4	3	0.05	0.20
d_5	2	0.10	0.30
Total			2.75

OPBST: Recurrence

Let $w(i, j) = \sum_{l=i}^j p_l + \sum_{l=i-1}^{j-1} q_l$ for $1 \leq i \leq j \leq n$.

Let $e(i, j)$ = expected cost of searching an optimal binary search tree containing the keys k_i, \dots, k_j .

Then $e(1, n)$ = expected cost of searching an optimal binary search tree containing k_1, \dots, k_n (i.e., containing all keys).

If k_r is the root of an optimal subtree containing k_i, \dots, k_j , then

$$\begin{aligned} e(i, j) &= p_r + \{e(i, r - 1) + w(i, r - 1)\} \\ &\quad + \{e(r + 1, j) + w(r + 1, j)\} \\ &= e(i, r - 1) + e(r + 1, j) + w(i, j) \end{aligned}$$

Hence,

$$e(i, j) = \begin{cases} q_{i-1}, & \text{if } j = i - 1, \\ \min_{i \leq r \leq j} \{e(i, r - 1) + e(r + 1, j) + w(i, j)\}, & \text{if } i < j. \end{cases}$$

OPBST: Bottom-up DP (Cubic Time)

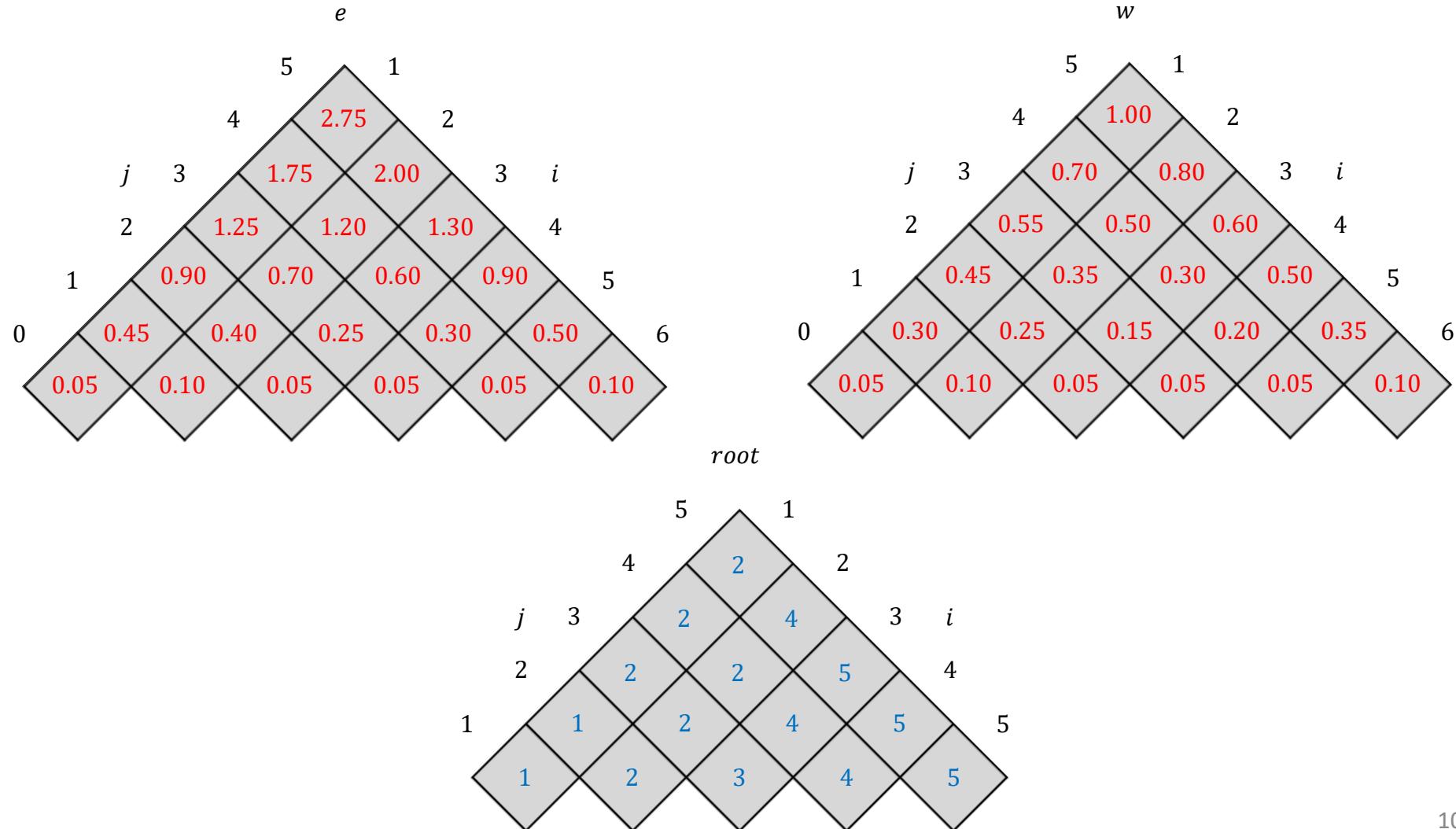
OPTIMAL-BST (p, q, n)

1. $e[1..n + 1, 0..n] \leftarrow$ new table,
- $w[1..n + 1, 0..n] \leftarrow$ new table,
- $root[1..n, 1..n] \leftarrow$ new table
2. **for** $i \leftarrow 1$ **to** $n + 1$ **do**
3. $e[i, i - 1] \leftarrow q_{i-1}$
4. $w[i, i - 1] \leftarrow q_{i-1}$
5. **for** $l \leftarrow 1$ **to** n **do**
6. **for** $i \leftarrow 1$ **to** $n - l + 1$ **do**
7. $j \leftarrow i + l - 1$
8. $e[i, j] \leftarrow \infty$
9. $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$
10. **for** $r \leftarrow i$ **to** j **do**
11. $t \triangleq e[i, r - 1] + e[r + 1, j] + w[i, j]$
12. **if** $t < e[i, j]$ **then**
13. $e[i, j] \leftarrow t$
14. $root[i, j] \leftarrow r$
15. **return** e **and** $root$

Running time = $\Theta(n^3)$

OPBST: Bottom-up DP (Cubic Time)

k_i	k_1	k_2	k_3	k_4	k_5	
p_i	0.15	0.10	0.05	0.10	0.20	
d_i	d_0	d_1	d_2	d_3	d_4	d_5
q_i	0.05	0.10	0.05	0.05	0.05	0.10



OPBST: Bottom-up DP (Quadratic Time)

OPTIMAL-BST (p, q, n)

1. $e[1..n + 1, 0..n] \leftarrow$ new table,
- $w[1..n + 1, 0..n] \leftarrow$ new table,
- $root[1..n, 1..n] \leftarrow$ new table
2. **for** $i \leftarrow 1$ **to** $n + 1$ **do**
3. $e[i, i - 1] \leftarrow q_{i-1}$
4. $w[i, i - 1] \leftarrow q_{i-1}$
5. **for** $l \leftarrow 1$ **to** n **do**
6. **for** $i \leftarrow 1$ **to** $n - l + 1$ **do**
7. $j \leftarrow i + l - 1$
8. $e[i, j] \leftarrow \infty$
9. $w[i, j] \leftarrow w[i, j - 1] + p_j + q_j$
10. **for** $r \leftarrow root[i, j - 1]$ **to** $root[i + 1, j]$ **do**
11. $t \triangleq e[i, r - 1] + e[r + 1, j] + w[i, j]$
12. **if** $t < e[i, j]$ **then**
13. $e[i, j] \leftarrow t$
14. $root[i, j] \leftarrow r$
15. **return** e **and** $root$

Running time = $\Theta(n^2)$

Optional

DP using Recursive Divide-and-Conquer

Rod Cutting: Recursive Divide-&-Conquer

DIVIDE-AND-CONQUER-CUT-ROD (p, n)

1. $r[0..n] \leftarrow$ new array
2. $r[0] \leftarrow 0$
3. *for* $i \leftarrow 1$ *to* n *do*
4. $r[i] \leftarrow -\infty$
5. *DC-CUT-ROD-A ($p, r, 1, n$)*
6. *return* $r[n]$

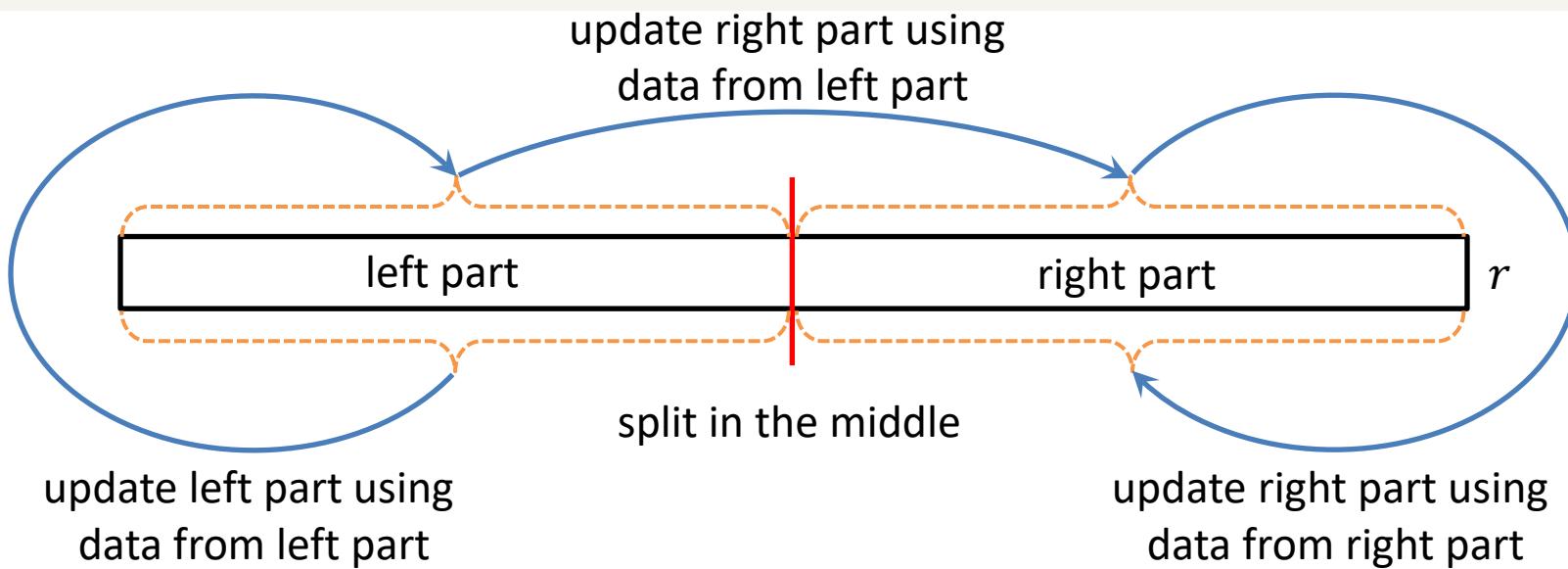
DC-CUT-ROD-SOLVE-BASE (p, r, k_1, n_1, k_2, n_2)

1. *for* $j \leftarrow k_2$ *to* $k_2 + n_2 - 1$ *do*
2. $q \leftarrow r[j]$
3. *for* $i \leftarrow k_1$ *to* $\min\{j, k_1 + n_1 - 1\}$ *do*
4. $q \leftarrow \max\{ q, p[i] + r[j-i] \}$
5. $r[j] \leftarrow q$

Rod Cutting: Recursive Divide-&-Conquer

DC-CUT-ROD-A (p, r, k, n)

1. *if* $n \leq \text{BASE_SIZE}$ *then*
2. *DC-CUT-ROD-SOLVE-BASE (p, r, k, n, k, n)*
3. *else*
4. $m \leftarrow \lfloor n/2 \rfloor$
5. *DC-CUT-ROD-A (p, r, k, m)* // update left part using left part
6. *DC-CUT-ROD-B ($p, r, k, m, k + m, n - m$)* // update right part using left part
7. *DC-CUT-ROD-A ($p, r, k + m, n - m$)* // update right part using right part



Rod Cutting: Recursive Divide-&-Conquer

DC-CUT-ROD-B (p, r, k_1, n_1, k_2, n_2)

1. *if* $n \leq \text{BASE_SIZE}$ *then*
2. *DC-CUT-ROD-SOLVE-BASE (p, r, k_1, n_1, k_2, n_2)*
3. *else*
4. $m_1 \leftarrow \lfloor n_1/2 \rfloor, m_2 \leftarrow \lfloor n_2/2 \rfloor$ *// let $L \equiv [k_1..k_1 + n_1 - 1]$ and $R \equiv [k_2..k_2 + n_2 - 1]$*
5. *DC-CUT-ROD-B (p, r, k_1, m_1, k_2, m_2)* *// left of L updates left of R*
6. *DC-CUT-ROD-B ($p, r, k_1 + m_1, n_1 - m_1, k_2, m_2$)* *// right of L updates left of R*
7. *DC-CUT-ROD-B ($p, r, k_1, m_1, k_2 + m_2, n_2 - m_2$)* *// left of L updates right of R*
8. *DC-CUT-ROD-B ($p, r, k_1 + m_1, n_1 - m_1, k_2 + m_2, n_2 - m_2$)* *// right of L updates right of R*

split in the middle

left part

right part

$L \equiv r[k_1..k_1 + n_1 - 1]$

left part

right part

$R \equiv r[k_2..k_2 + n_2 - 1]$

Rod Cutting: Recursive Divide-&-Conquer

Let $T(n)$, $T_A(n)$ and $T_B(n)$ be the running times of *DIVIDE-AND-CONQUER-CUT-ROD*, *DC-CUT-ROD-A* and *DC-CUT-ROD-B*, respectively, on an input of size n . Then

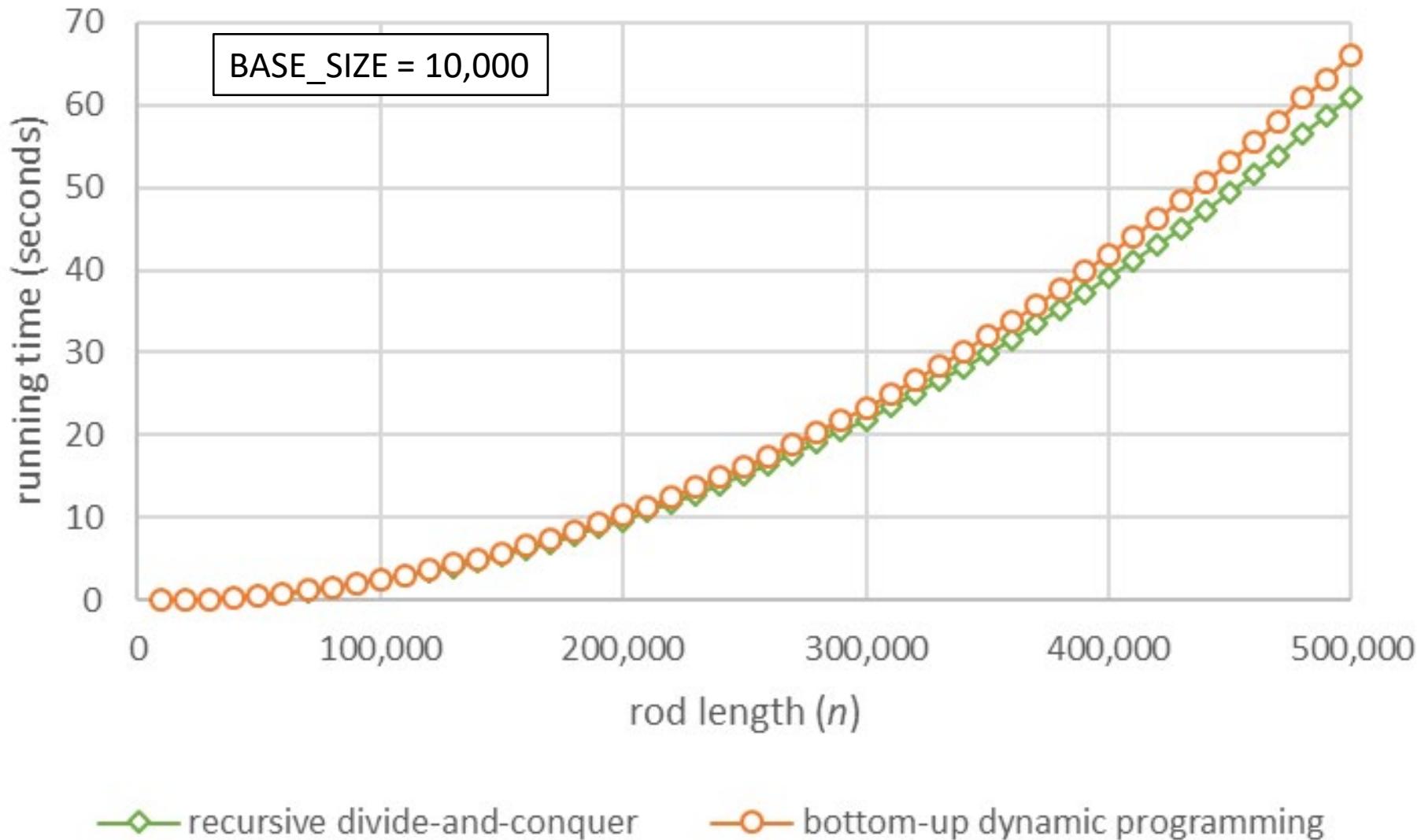
$$T(n) = T_A(n) + \Theta(n).$$

$$T_A(n) = \begin{cases} \Theta(1), & \text{if } n \leq \text{BASE_SIZE}, \\ 2T_A\left(\frac{n}{2}\right) + T_B\left(\frac{n}{2}\right) + \Theta(1), & \text{otherwise.} \end{cases}$$

$$T_B(n) = \begin{cases} \Theta(1), & \text{if } n \leq \text{BASE_SIZE}, \\ 4T_B\left(\frac{n}{2}\right) + \Theta(1), & \text{otherwise.} \end{cases}$$

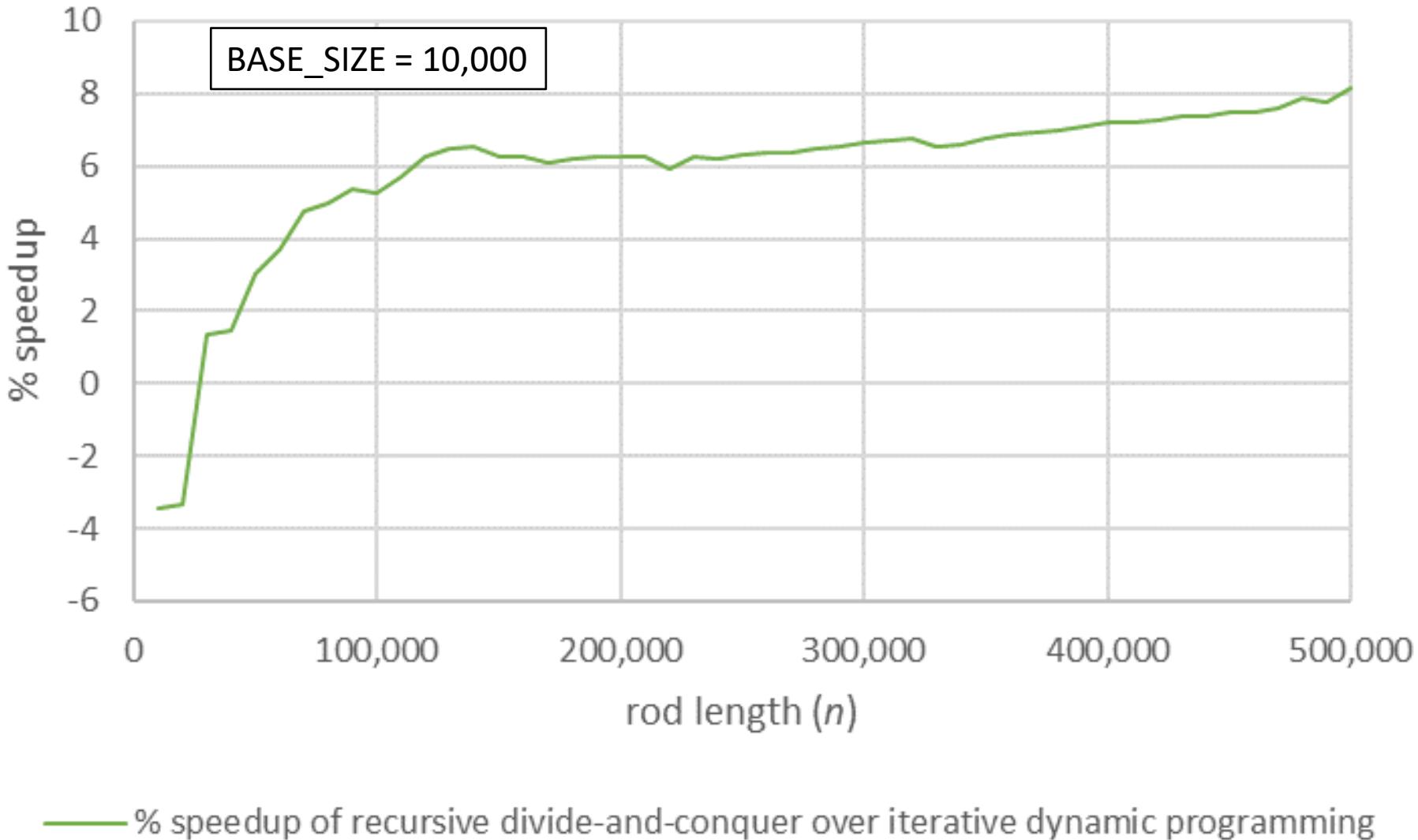
Solving: $T(n) = \Theta(n^2)$.

Rod Cutting: Recursive Divide-&-Conquer



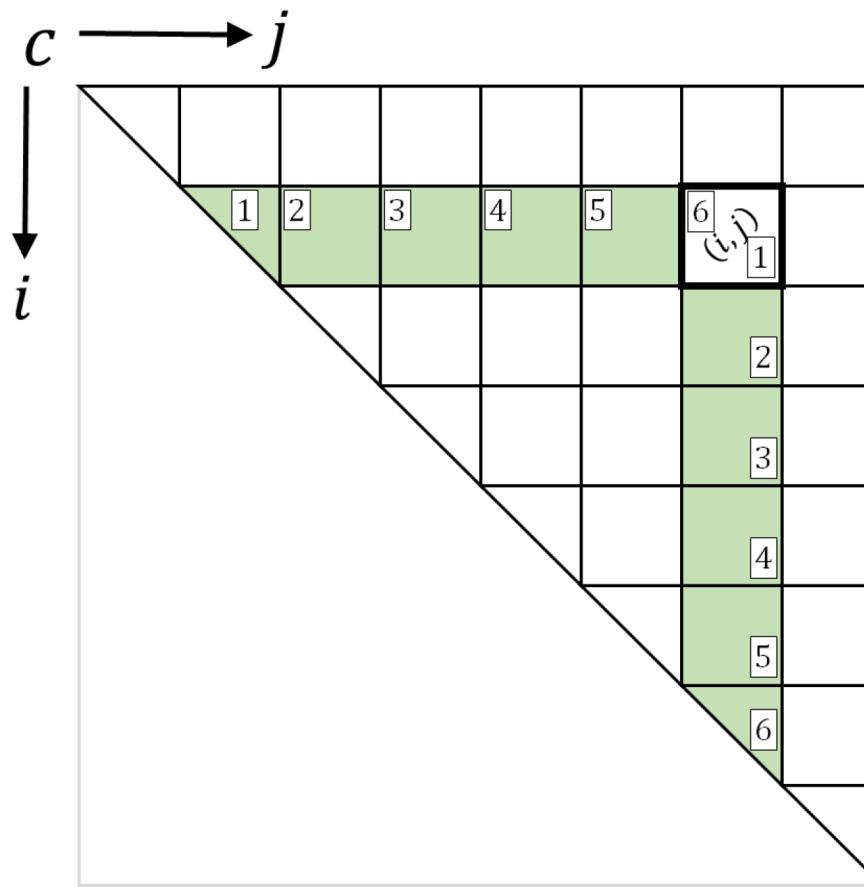
*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Rod Cutting: Recursive Divide-&-Conquer

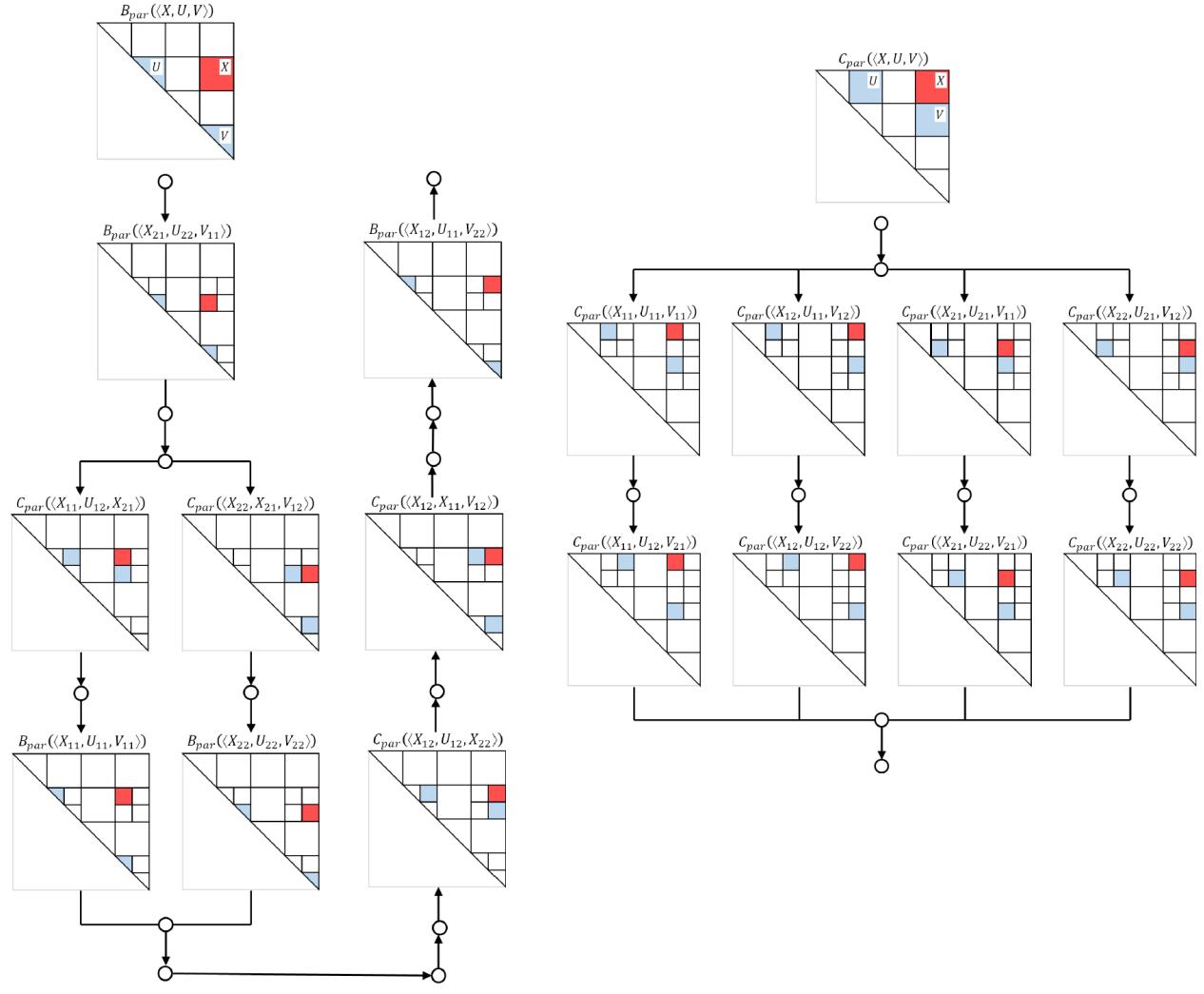
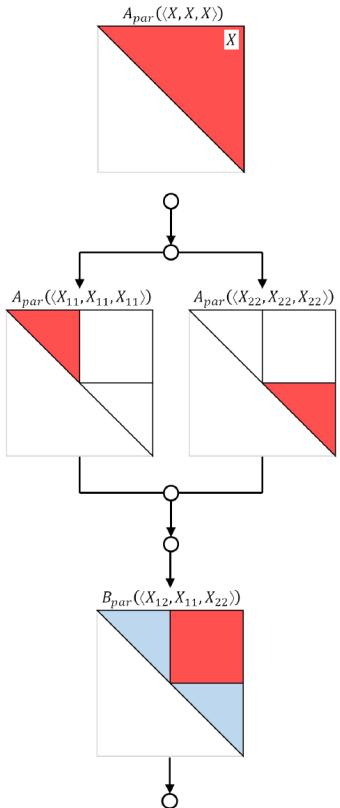
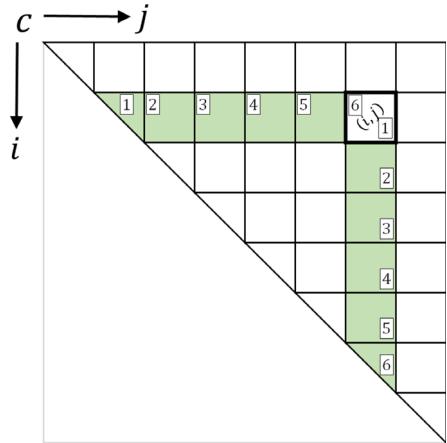


*Run on a dual-socket (2 × 8 cores) 2.0 GHz Intel E5-2650 with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM. Only one core was used.

Matrix-Chain Mult: Recursive Divide-&-Conquer



Matrix-Chain Mult: Recursive Divide-&-Conquer



Matrix-Chain Mult: Recursive Divide-&-Conquer

$\mathcal{A}_{par}(\langle X, X, X \rangle)$

1. **if** X is a small matrix **then** $\mathcal{A}_{loop-par}(\langle X, X, X \rangle)$
2. **else**
3. **par:** $\mathcal{A}_{par}(\langle X_{11}, X_{11}, X_{11} \rangle), \mathcal{A}_{par}(\langle X_{22}, X_{22}, X_{22} \rangle)$
4. $\mathcal{B}_{par}(\langle X_{12}, X_{11}, X_{22} \rangle)$

$\mathcal{B}_{par}(\langle X, U, V \rangle)$

1. **if** X is a small matrix **then** $\mathcal{B}_{loop-par}(\langle X, U, V \rangle)$
2. **else**
3. $\mathcal{B}_{par}(\langle X_{21}, U_{22}, V_{11} \rangle)$
4. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle), \mathcal{C}_{par}(\langle X_{22}, X_{21}, V_{12} \rangle)$
5. **par:** $\mathcal{B}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle), \mathcal{B}_{par}(\langle X_{22}, X_{22}, V_{22} \rangle)$
6. $\mathcal{C}_{par}(\langle X_{12}, U_{12}, X_{22} \rangle)$
7. $\mathcal{C}_{par}(\langle X_{12}, X_{11}, V_{12} \rangle)$
8. $\mathcal{B}_{par}(\langle X_{12}, U_{11}, V_{22} \rangle)$

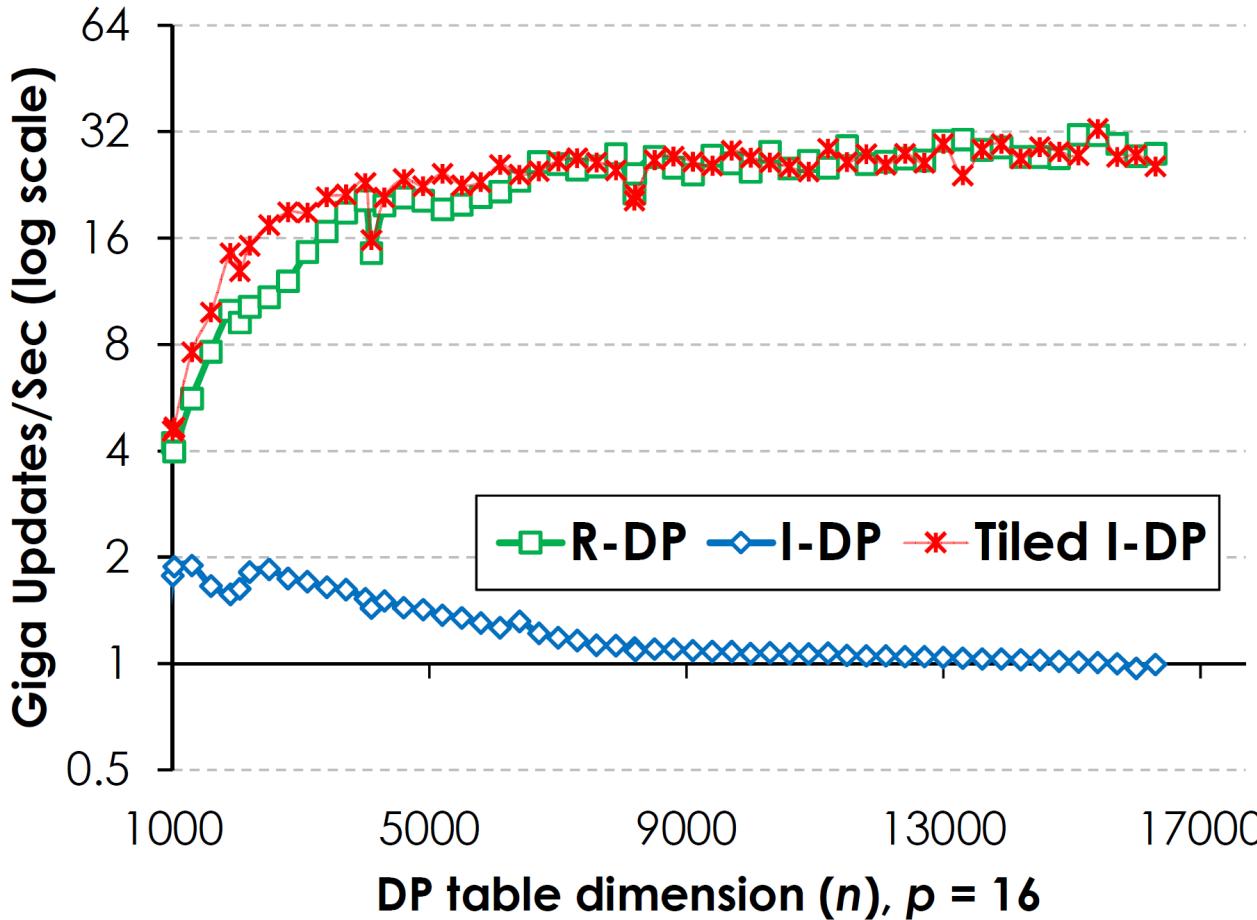
$\mathcal{C}_{par}(\langle X, U, V \rangle)$

1. **if** X is a small matrix **then** $\mathcal{C}_{loop-par}(\langle X, U, V \rangle)$
2. **else**
3. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{11}, V_{11} \rangle), \mathcal{C}_{par}(\langle X_{12}, U_{11}, V_{12} \rangle),$
 $\mathcal{C}_{par}(\langle X_{21}, U_{21}, V_{11} \rangle), \mathcal{C}_{par}(\langle X_{22}, U_{21}, V_{12} \rangle)$
4. **par:** $\mathcal{C}_{par}(\langle X_{11}, U_{12}, V_{21} \rangle), \mathcal{C}_{par}(\langle X_{12}, U_{12}, V_{22} \rangle),$
 $\mathcal{C}_{par}(\langle X_{21}, U_{22}, V_{21} \rangle), \mathcal{C}_{par}(\langle X_{22}, U_{22}, V_{22} \rangle)$

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

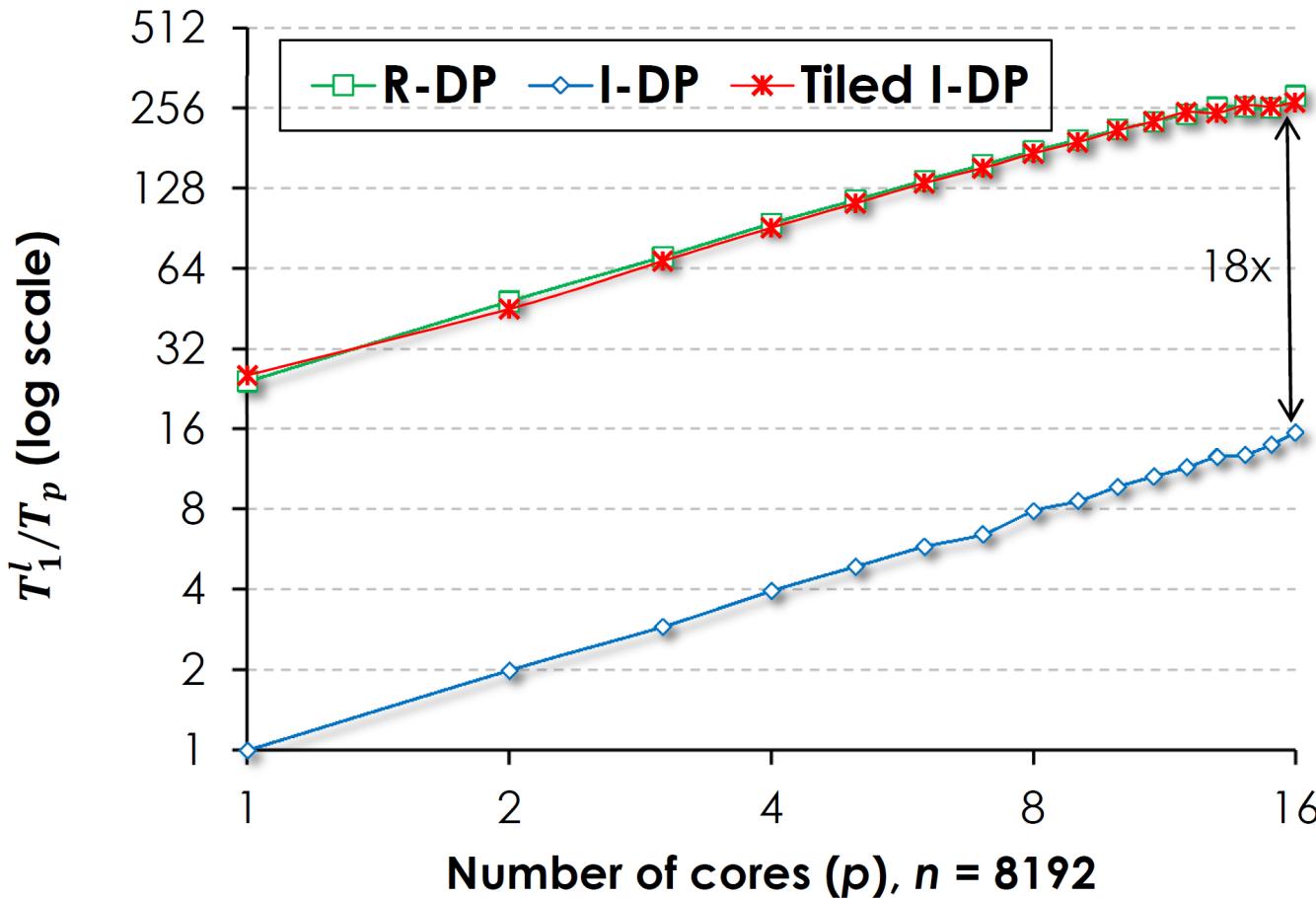


*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

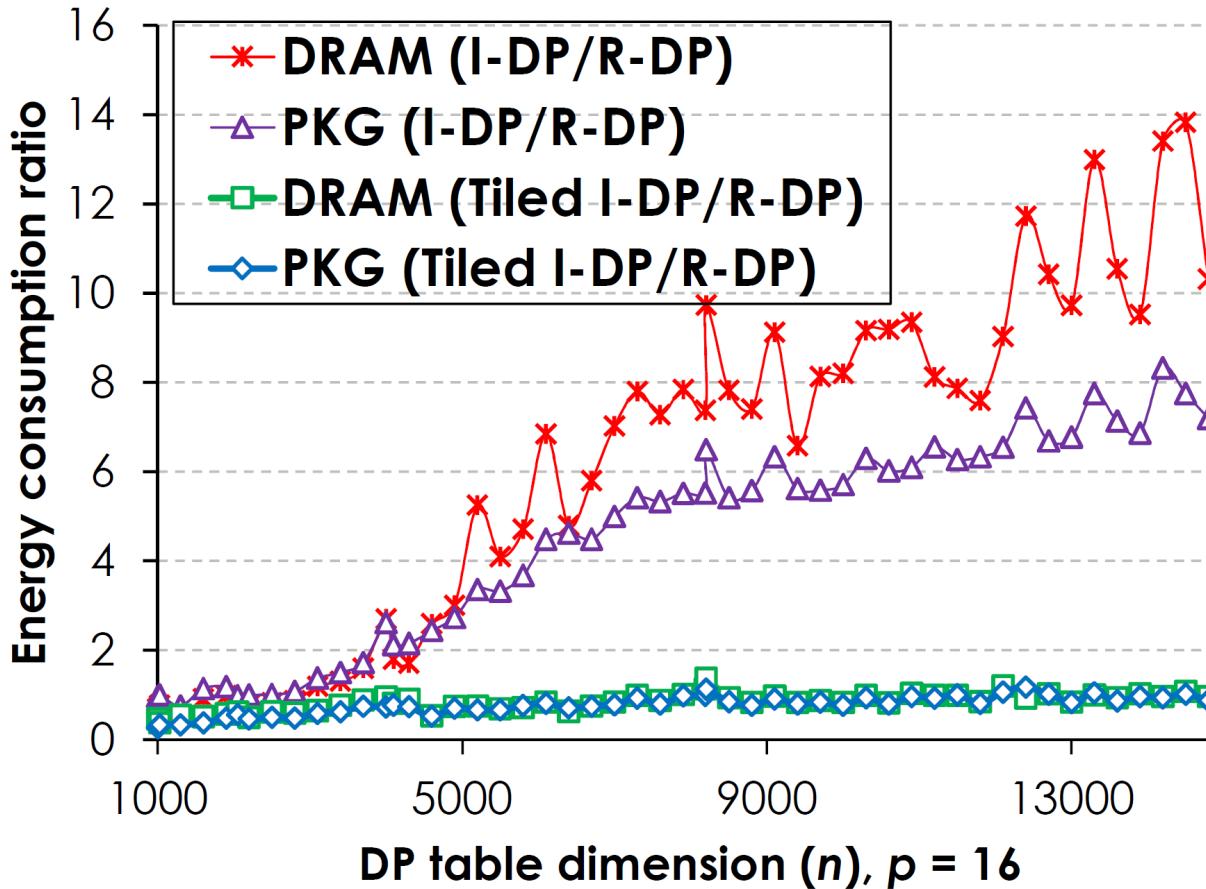


*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

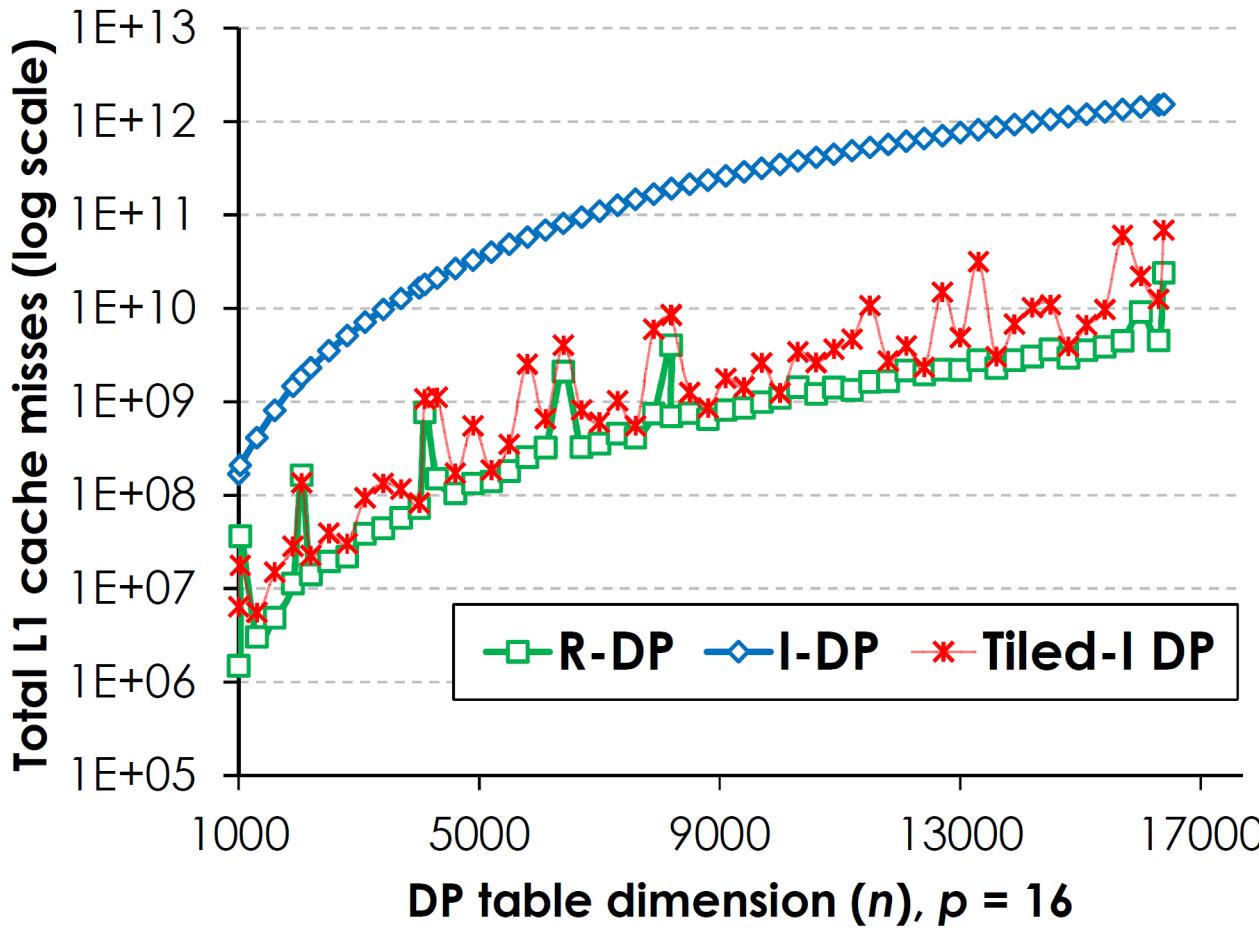


*Run on a dual-socket (2×8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

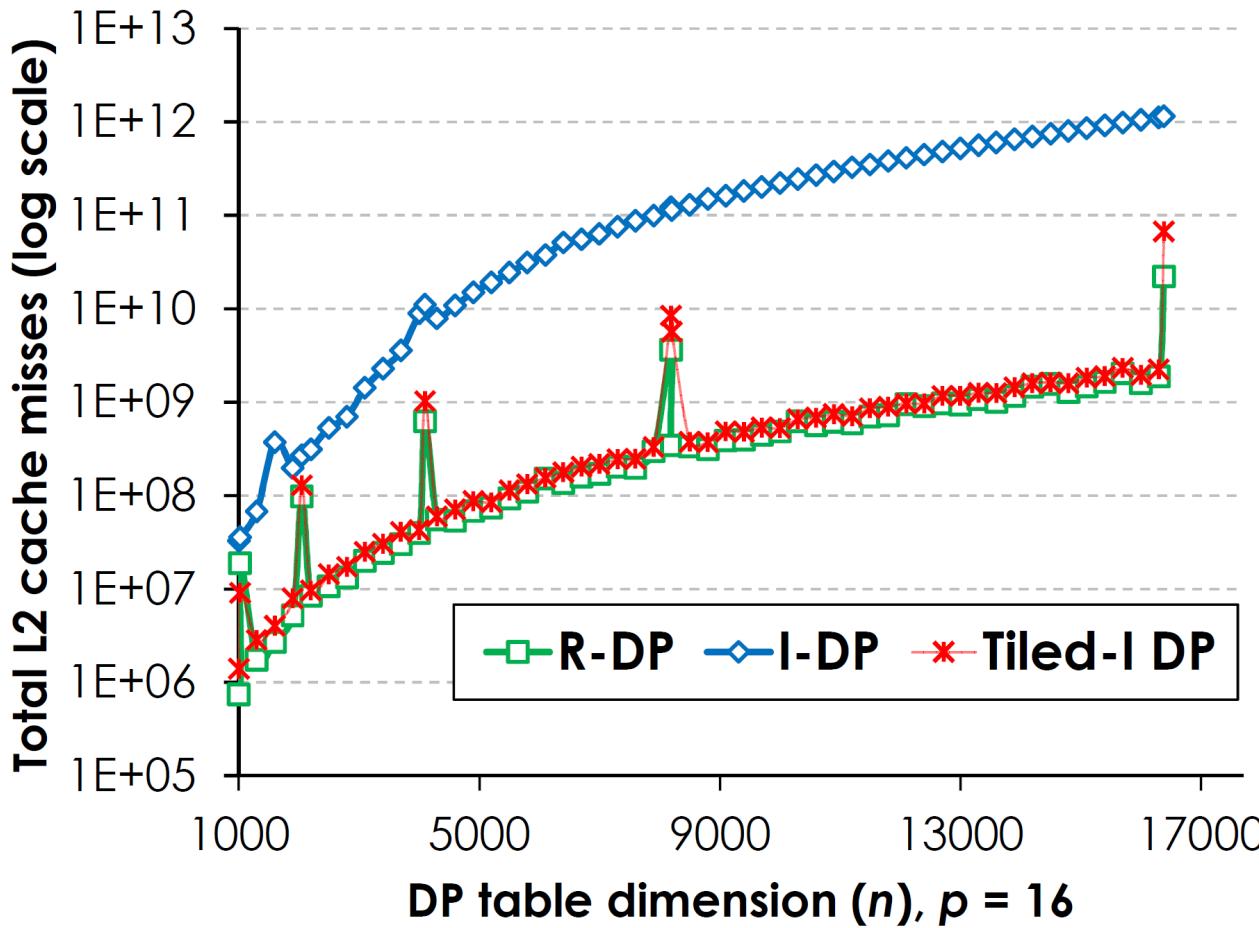


*Run on a dual-socket (2 × 8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

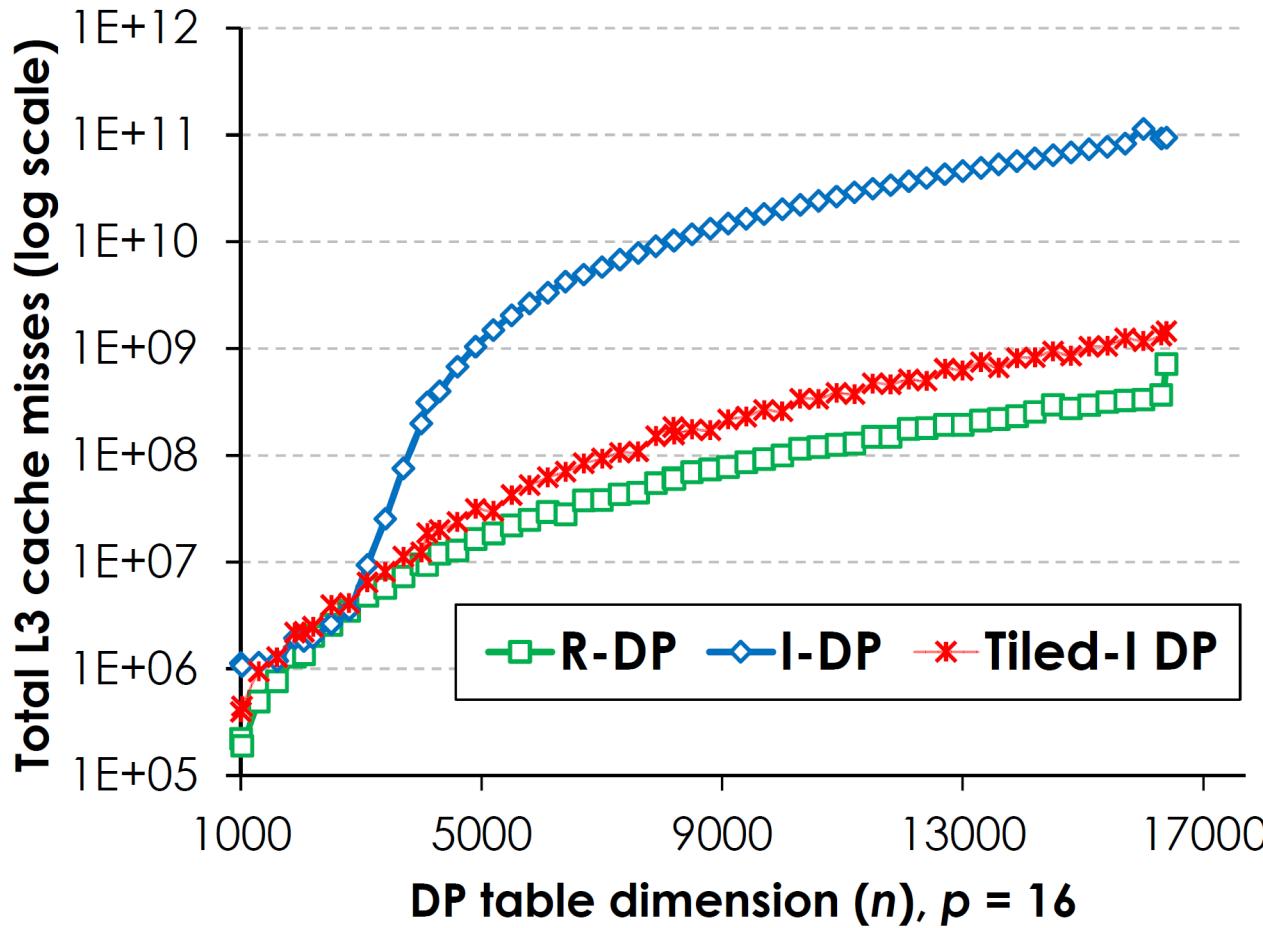


*Run on a dual-socket (2 × 8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, Tiled I-DP: tiled iterative DP

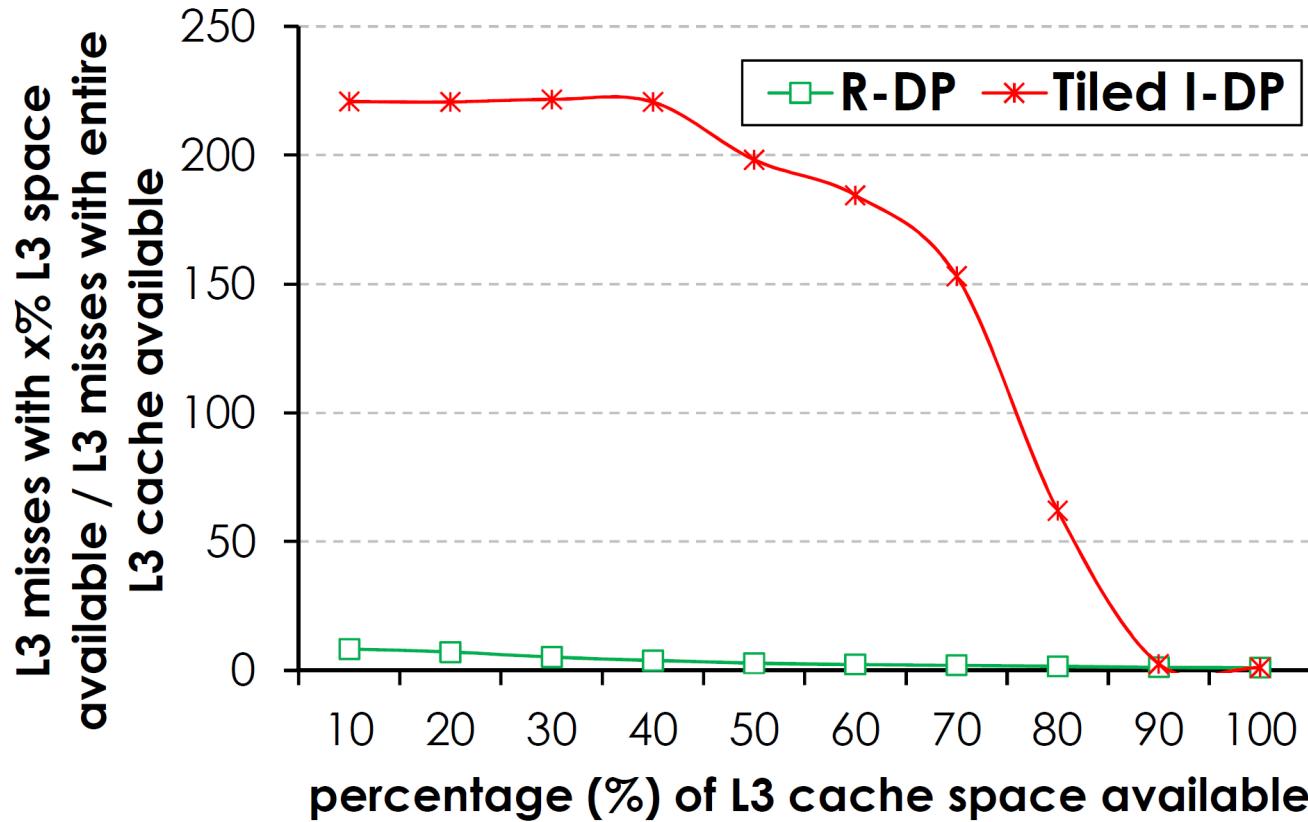


*Run on a dual-socket (2×8 cores) 2.7 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, **Tiled I-DP**: tiled iterative DP

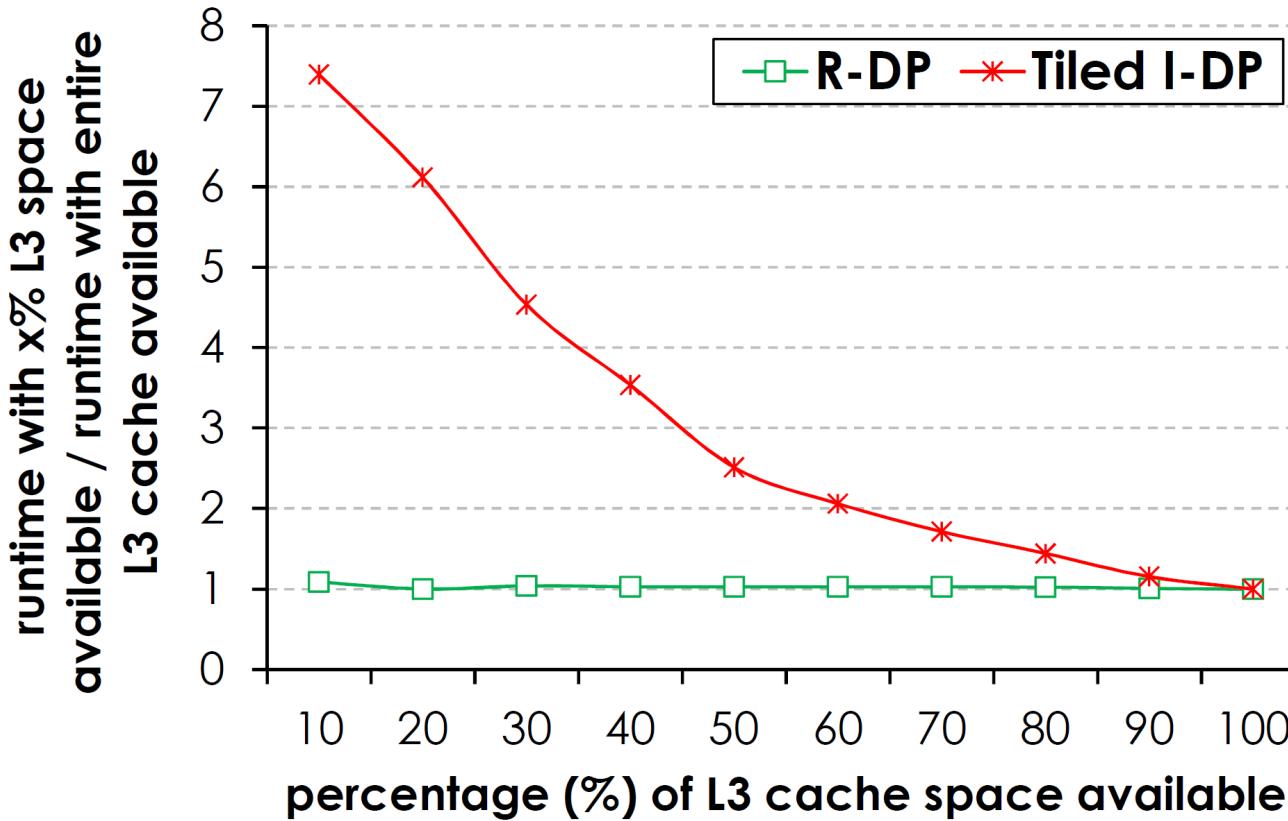


*Run on a dual-socket (2×8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

Matrix-Chain Mult: Empirical Performance

R-DP: recursive divide-&-conquer (BASE_SIZE = 64×64),

I-DP: iterative DP, **Tiled I-DP**: tiled iterative DP

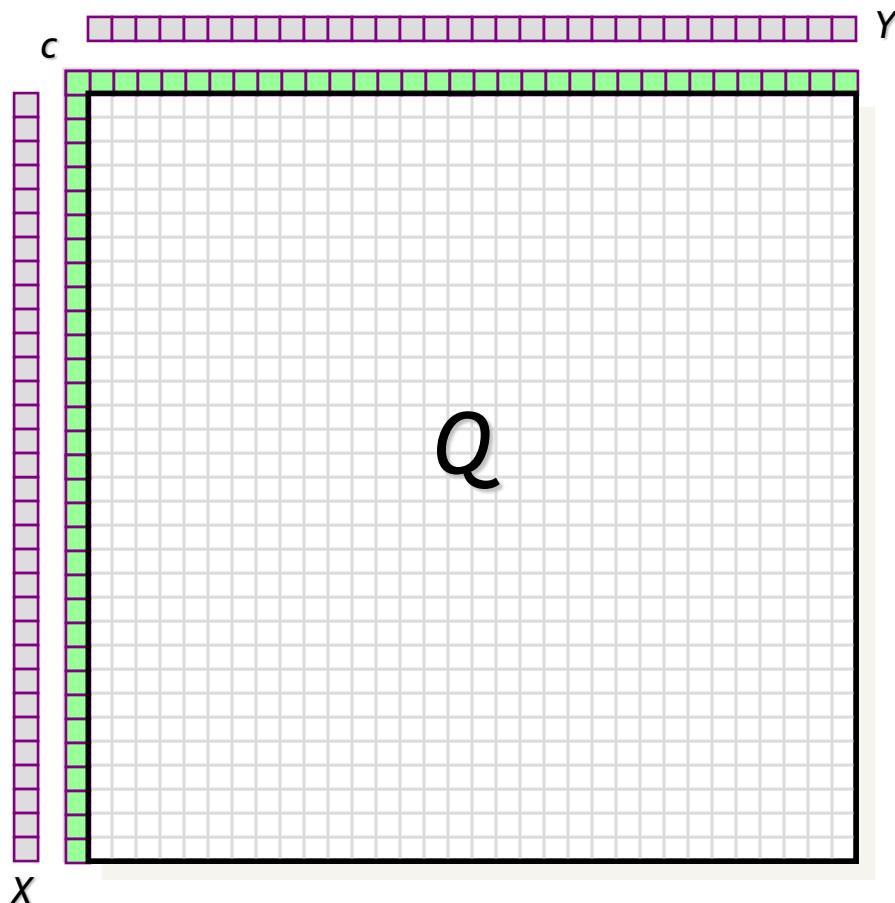


*Run on a dual-socket (2×8 cores) 2.0 GHz Intel Sandy Bridge with private 32KB L1 and 256KB L2 caches, a shared 20MB L3 cache per socket and 32GB RAM.

LCS: Linear Space with Recursive Divide-&-Conquer

$$Q \equiv c[1 \dots n, 1 \dots n]$$

$$\underline{n = 2^q}$$



■ stored values

LCS: Linear Space with Recursive Divide-&-Conquer

$$Q \equiv c[1 \dots n, 1 \dots n]$$

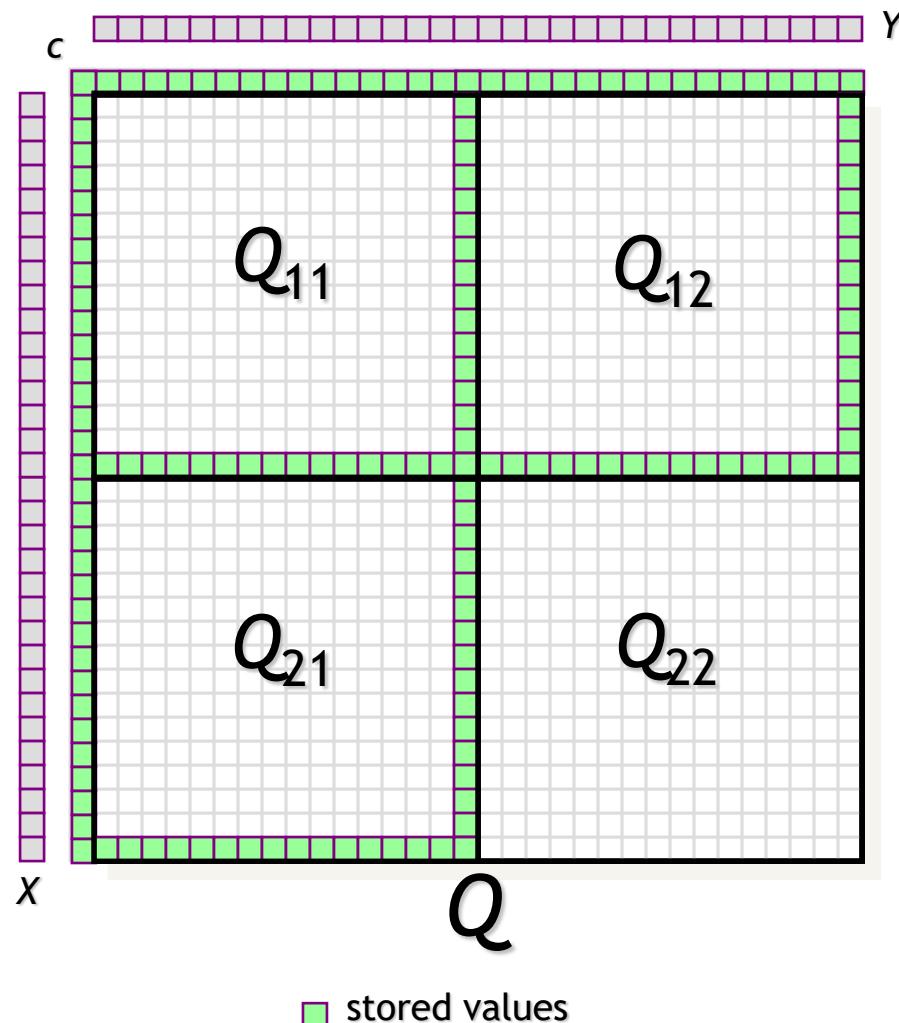
$$\underline{n = 2^q}$$

1. Decompose Q:

Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):

Generate the right and the bottom boundaries of the quadrants recursively.
(of at most 3 quadrants)



LCS: Linear Space with Recursive Divide-&-Conquer

$$Q \equiv c[1 \dots n, 1 \dots n]$$

$$n = 2^q$$

1. Decompose Q:

Split Q into four quadrants.

2. Forward Pass (Generate Boundaries):

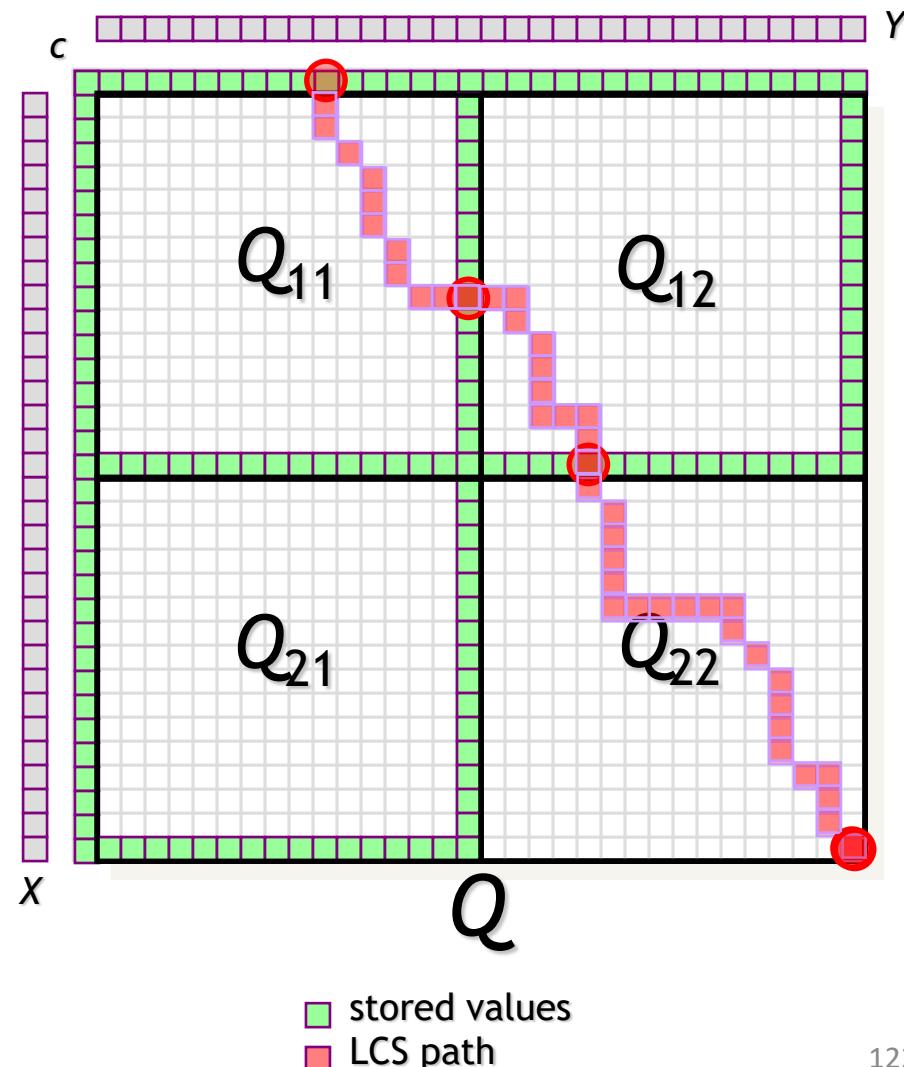
Generate the right and the bottom boundaries of the quadrants recursively.
(of at most 3 quadrants)

3. Backward Pass (Extract LCS-Path Fragments):

Extract LCS-Path fragments from the quadrants recursively.
(from at most 3 quadrants)

4. Compose LCS-Path:

Combine the LCS-Path fragments.



■ stored values
■ LCS path